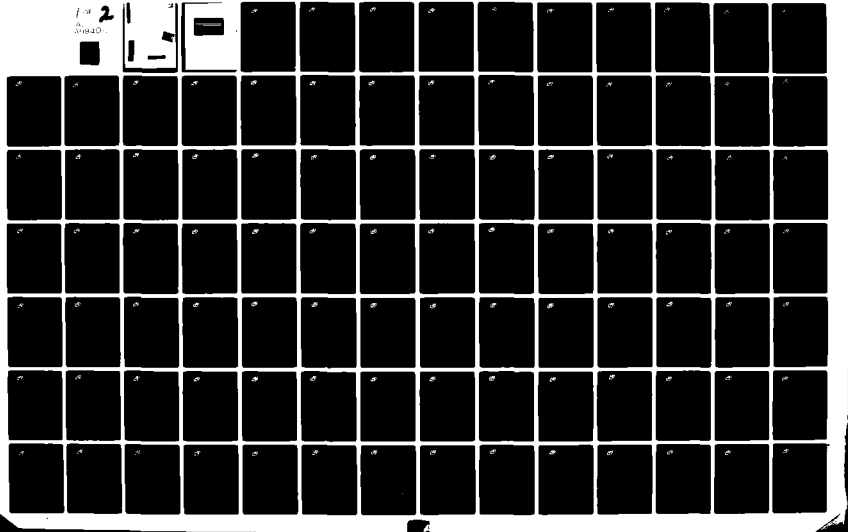
AD-A119 405 FAIL-SAFE TECHNOLOGY CORP LOS ANGELES CA F/G 9/2
U.S. NAVY FAULT-TOLERANT MICROCOMPUTER.(U)
JUL 82 M W SIEVERS, G A KRAVETZ, B A DUSSIA N00014-82-C-0126
UNCLASSIFIED NL



AD A119405

DTIC
ELECTE
SEP 2 0 1982
S D

DTIC FILE COPY

DISTRIBUTION STATEMENT
Approved for public release;
Distribution Unlimited

FAIL-SAFE TECHNOLOGY CORP. 008
SUITE 105
8929 SEPULVEDA BLVD.
LOS ANGELES, CALIFORNIA 90045

[redacted]

To: [redacted]

DEFENSE TECHNICAL INFORMATION CENTER

(12)

Fail-Safe Technology Corporation

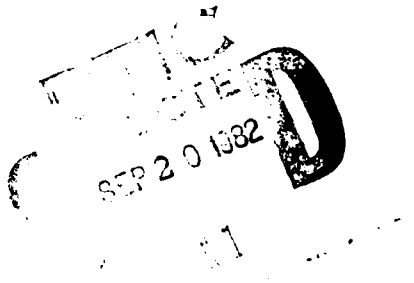
MILITARY STANDARD FAULT-TOLERANT MICROCOMPUTER

CONTRACT NUMBER N00014-82-C-0126

Final Report 0001AD

Prepared for

Office of Naval Research
Department of the Navy
800 N. Quincy Street
Arlington, Virginia 22217



Prepared by

Dr. M. W. Sievers
G. A. Kravetz
B. A. Dussia
J. D. Jackson

July 1982

This document has been approved for public release and sale; its distribution is unlimited.

Suite 106 • 8929 Sepulveda Boulevard • Los Angeles, California 90045 • Telephone: 213 / 417-4988

ABSTRACT

This Fail-Safe Technology Report includes the results of a feasibility study, preliminary design and recommendations for subsequent work. Fault tolerance is the unique attribute of a computer system that enables that system to continue its program-specified behavior in spite of the occurrence of faults. This feature is essential and cost-effective in applications where computer failure causes human injury or loss of critical data or equipment. For less critical applications, inclusion of fault detection and diagnostic procedures within the computer itself greatly increases repair time with a concomitant reduction in maintenance cost and downtime.

The computer system described in this report is constructed from off-the-shelf building block modules configured in a redundant manner to provide fault detection, isolation, and repair. The architectural features of the suggested FST system are: multiple loosely coupled processors executing identical tasks, redundant bus structure and self-checking Input-Output Controllers that vote on the outputs from the processors. Self-checking circuitry is used to resolve the question of "who checks the checker."

Included in this report are discussions of the applications, tradeoffs and requirements that led to the suggested architecture. An Executive Summary in Section 3 outlines the results of these discussions. The preliminary suggested hardware and software design is then shown. Specific recommendations for direction of further productive efforts are also included in this report. An overview of the state-of-the-art concepts and techniques employed in fault tolerant computer designs is added as Appendix I.

DTIC COPY INSPECTED 2

Accession For
NTIS GRA&I
DTIC TAB
Unannounced
Justification
By
Distribution
Availability Codes
Dist | Avail and/or Special
A

TABLE OF CONTENTS

LIST OF FIGURES

TABLES

1.0 SCOPE

This report contains the results of a feasibility study, preliminary hardware and software architecture design and recommended future work for a Naval Standard Fault-Tolerant Computer (NSFTC) system. An appendix surveys the state-of-the-art in fault tolerant computer design and analysis. A proposal to develop and implement a demonstration system that exhibits the essential characteristics of the NSFTC will follow the publication of this document.

2.0 CONTRACT ACKNOWLEDGEMENT

This report is the result of a contract granted under the Defense Small Business Advanced Technology Program (DESAT) to study the feasibility of fault-tolerant computer design for the Navy. This study was funded for six months under Office of Naval Research contract number N00014-82-C-0126. Technical management was provided by NORDA in Mississippi. Fail-Safe Technology wishes to thank the many Navy personnel contacted during the course of this work for their support and willingness to discuss potential applications for fault tolerant computers.

3.0 EXECUTIVE SUMMARY

Modern Naval computer usage, in combat systems, aerospace applications or routine shipboard process control, requires the highest level of availability, reliability and maintainability. Computer errors at any significant level can be disastrous in terms of human injury, aborted missions, loss of critical information and equipment damage. Moreover, lengthy computer down-time can result in the occurrence of major cost overruns, while computer-dependent resources are out of service and maintenance personnel are brought in from distant points.

Two expensive methods have been traditionally used to ensure an acceptable level of operational readiness. One involves stocking a large quantity of spare components in on-site facilities and deploying highly skilled maintenance teams with each computer installation. These resources are seldom used, but must be available on very short notice in the event of a computer failure. The second solution is to deploy many more systems than necessary so that there is always a high probability that a required number are functional at all times. Both solutions are costly in both man-hours and equipment duplication; neither affords increased effectiveness. Now there is a third alternative available.

3.1 A Fault-Tolerant Solution: Reduced Costs and Increased Effectiveness

Fail-Safe Technology is a company that specializes in offering a practical new solution to the computer error and maintenance

problem. This solution makes optimum use of computer resources while offering major long-term savings over the previous generation of computer systems. Fault-tolerant computer systems are capable of monitoring their own functions and automatically initiating repair when a fault is detected. The calculations quoted in the following chapters suggest that fault tolerance can reduce costs over the life of a system from 500 percent for small systems to over 6000 percent for large systems.

Self-maintaining fault-tolerant systems are attractive for a number of reasons:

1) Self-maintenance allows the computer to complete its mission safely at a rate approaching 100% certainty.

2) Self-maintenance puts an end to the need for deployment of highly skilled maintenance teams or spare components with each computer.

3) Any necessary repair can be delayed until it can be conveniently performed, for example, when a ship has returned to port. This feature permits a highly effective centralized supply and repair facility.

3.2 Summary of System Considerations

Within the normal life cycle of most computer systems in use today, maintenance costs will ultimately exceed purchase price. Fault tolerant systems reduce ongoing costs of both normally scheduled preventive maintenance and corrective maintenance of system failures. Section 4 includes a Naval applications survey which covers major computer system use on shipboard, in buoys,

aboard aircraft or satellites and in combat coordination. All of these applications require a low incidence of outages and all could become far more cost-effective with fault tolerant design.

This survey indicates that a fault-tolerant micro-computer system tailored to perform real-time process control is suitable in the widest range of applications. The locus of requirements for such a computer may be summarized as:

1) amenable to the use of high level languages
2) no permanent corruption of stored data
3) no interruption in service for periods of hours or years (depending on the application)
4) automatic maintenance procedures to shorten manual repair time
5) no erroneous outputs may be generated at any time (for certain highly critical applications)

3.3 Feasibility Analysis

Section 5 is a detailed analysis of reliability, availability and maintainability of simplex and fault-tolerant computer systems. This analysis examines low, medium and high performance computer systems and indicates that the greatest improvements in these parameters are achieved in the more complex systems. Another important result of this analysis indicates that external sparing (i.e., replacing entire failed computers) requires many costly spares to achieve high reliability. Internal sparing (i.e., providing internal redundancy to improve unit reliability) great-

ly reduces the cost of high reliability.

The most significant result of this analysis is the reduction of life-cycle costs achieved by fault-tolerant computing. In this analysis life-cycle costs are estimated by computing the cost per operating hour for preventive and corrective maintenance procedures. As previously stated, for the simplified calculations made, fault tolerance can reduce life-cycle costs by 500 percent for small systems to over 6000 percent for larger systems. This cost reduction compensates for the added initial expense of providing fault tolerance several times over and continuously avoids other possible costs.

3.4 NSFTC Theory of Operation

The requirements listed in Section 4 have led to the design of a highly flexible distributed computer architecture which Fail-Safe Technology has designated the Naval Standard Fault-Tolerant Computer (NSFTC). The fault-tolerant architecture of this computer offers a hierarchical fault detection mechanism and a simple recovery technique. Fault detection is organized to recognize errors at several different levels so that higher level checking detects errors missed at lower levels.

When a fault is detected, recovery can, in general, take several forms. In the NSFTC, recovery is based on the use of a virtual resource allocation method termed soft-naming. The system can

nullify the soft name of a failed resource and then assign the name to a spare. The spare then operates in place of the nullified resource.

3.5 Hardware Design and System Software

The hardware for the NSFTC is based on a redundant configuration of loosely coupled non-redundant off-the-shelf single board or small "black-box" processors (SBPs), a redundant bus architecture and self-checking Input-Output Controllers (IOCs). SBPs can be existing or future Navy computers. The SBPs independently execute identical tasks and periodically exchange results for error-checking purposes. Output data are transmitted over the redundant bus to the self-checking IOC, which is a small special-purpose computer. Self-checking is a property of a circuit that enables that circuit to check itself for faults while operating normally. These circuits are employed to resolve the question "who checks the checker?"

The IOC votes on information received from the bus and outputs the majority decision. Thus no single failure will result in a corrupted output. The IOC similarly checks the validity of input data and distributes this information to the redundant bus. In the event of a permanent IOC fault, the IOC will remove itself from service, and its tasks are resumed by a spare IOC.

The IOC and a Bus Adaptor which connects the several processors to the redundant bus are both small custom building block mod-

ules. The SBPs are off-the-shelf components. One of the most important aspects of this architecture is that the fault-tolerant design places virtually no restrictions on the SBP itself. Hence, this module may be selected on the basis of its suitability to the application task. This latter point is very important because it permits the use of any processing element. Applications requiring the use of a Military Standard Computer may still be made fault-tolerant by embedding this computer in the proposed architecture.

The distributed NSFTC operating system provides such normal services as resource sharing and task scheduling. In addition, it includes features necessary for redundancy management. The operating system will be designed to operate under three different conditions: normal operation; the interval between the occurrence of a fault and its detection; and the period between detection and recovery. In order to accommodate these states, the operating system comprises fixed components for normal periods as well as variable recovery software.

3.6 Phase II Recommendations

The Phase II recommendations concentrate on firming the concepts established in the Phase I study. This second phase will design, implement and test an advanced development model (ADM) NSFTC that exhibits the fundamental concepts of the fault-tolerant methodology. This demonstration is an essential precursor to the full

scale development (FSD) since it permits actual testing of the system functionality and fault-tolerance algorithms. Phase II will also include an expanded study of the requirements for specific Navy applications.

4.0 SYSTEM CONSIDERATIONS

Reliability is a problem of great concern in complex Naval systems. Unexpected failures are costly in many ways, including losses in operational readiness, the need for additional personnel to effect maintenance and a large number of spare systems. Due to the separation in responsibility between initial procurement and subsequent field service, it is generally difficult to quantify the actual dollar amounts involved in purchasing and owning a system. However, it can be safely postulated that the cost of maintaining a system is far greater than the cost of its initial purchase.

The effect of fault-tolerant computer design on the overall life-cycle cost of embedded-computer systems may be significant in a large number of applications. Life-cycle costs may be greatly reduced by designing increased testability and maintainability into the computer. Fault-tolerant computer systems can extend periods between scheduled maintenance and postpone corrective maintenance until it can be conveniently performed.

Preventive or scheduled maintenance comprises those procedures employed to reduce the likelihood of system failure. Such procedures generally require extensive system testing and replacement of high failure-rate components, whether or not their failure is imminent. Corrective maintenance is the philosophy in which service is initiated after the system has failed. Many

systems employ both techniques to increase overall availability.

Automatic maintenance procedures can make a significant impact on reducing costs of preventive and corrective maintenance. This is because automatic maintenance performs corrective procedures without assistance from service personnel. Preventive maintenance costs are also reduced, since the period between scheduled maintenance is increased. Furthermore, the system itself can assist service personnel during preventive maintenance thereby minimizing the maintenance time.

The starting point for automatic maintenance is the computers within the system. If the reliability of a computer is assured, the computer may be used to execute subsystem testing, fault diagnosis and spare switching. The technique described in this report involves slightly increasing the purchase cost of a system by adding an automatic fault-handling capability, in order to greatly reduce maintenance costs during the lifetime of the system. The technique employs redundant components organized in an architecture that can manage internal faults without human intervention. The following subsections deal with Naval applications, reliability requirements and the major functions of such a system.

4.1 Application Survey and Reliability Requirements

One of the first steps in the design of a computer system is to develop a set of requirements that must be satisfied by the

system. These requirements must specify performance, reliability and availability goals. The function of the computer designer is to tailor the architecture to meet the specified requirements. A small study was initiated as part of this effort to gauge the range of Naval applications that would benefit most from fault-tolerant computer technology. This study yielded a good insight into the common reliability and availability problems faced by the Navy, and generated a great deal of interest in the technology.

The Naval applications that could benefit most from the use of fault-tolerant computers are those associated with shipboard control and monitoring, buoys, aircraft functions, satellites and combat coordination systems. The common factor in all of these applications is the need for maintenance and diagnostic algorithms that are capable of quickly indicating the "least replaceable unit" to service personnel when scheduled maintenance is performed. In this subsection, typical Naval applications and reliability requirements for fault tolerant-computers are tabulated.

4.1.1 Shipboard Applications

Computer systems used onboard Naval vessels face a unique operational environment. Perhaps the most vexing problem to be encountered in dealing with shipboard computers is the large number of phenomena that potentially cause transient computer faults.

Conventional computer systems protect against power transients through power conditioning and battery backup. However, other transient fault mechanisms exist that are not related to the power supply. The fault-tolerant computer is capable of detecting and repairing the damage caused by random, localized transient faults of short duration. The NSFTC architecture proposed in this report has the unique capability of continuing operation during and after a transient fault without any noticeable external effects.

Automatic shipboard status monitoring and process control must be available 24 hours a day, 7 days a week, in and out of port. Although brief outages in service are permitted in these applications, it is paramount that memory be protected against permanent corruption. This application necessitates sophisticated off-line fault diagnosis procedures, in addition to automatic fault repair, in order for maintenance personnel to rapidly isolate and renew portions of the system that have failed.

Fire control computers are characterized by requirements for high availability and reliability for relatively short periods of time (typically less than two hours). Outages of any duration are not permissible and under no circumstance can the computer produce an unsafe output. Instantaneous fault masking is therefore required to prevent erroneous output. These general requirements and time scales are often best handled by a triple-modular-redundant (TMR)

computer configuration in which a majority vote is taken on the outputs from three processors. For short duration missions, this configuration yields very high reliability and meets the requirement of not propagating faulty results to the output.

Ocean survey applications often require computer equipment to operate without interruption for a minimum of 28 days. Short computer outages are permitted while reconfiguration is taking place after a failure. There are generally at least seven days in port after any mission for any needed repairs.

4.1.2 Buoys

Buoys currently contain data storage equipment in the form of highly ruggedized tape recorders. Due to the potentially dangerous and often harsh environments in which buoys may be placed, as well as the high cost that would be incurred under any circumstances, retrieving buoys for repair is not generally practicable. As a result, sophisticated computer intelligence has not been placed in these buoys because conventional computer systems are not reliable enough. However computers within buoys could be useful in such applications as reducing data which could then be collected by satellites. These computers would have to be small in volume and low in power consumption. Outages during fault recovery would be permitted should fault tolerant-computers be placed inside buoys.

4.1.3 Aircraft

Currently, the F/A-18 aircraft has a fault-tolerant flight control computer. There are other on-board systems that are computer-controlled, but are not protected against failures. In the event that one of these systems fails, the general philosophy is for the aircraft to return to the carrier without completing its mission. Typical mission duration is two to four hours, and short outages during recovery from a fault would become acceptable, once the non-redundant computer systems are replaced with fault-tolerant systems.

4.1.4 Satellites

The U.S. Navy makes use of satellites for various applications. These satellites may contain sophisticated controllers and signal processing equipment. Although designing fault tolerance into signal processors is extremely complex and costly, on-board control computers are very amenable to this technology.

4.1.5 Combat Coordination

NOSC in San Diego is considering the development of a sophisticated computer-controlled combat coordination system. This system would monitor several essential parameters and assist in development of combat strategy. A large database is maintained that contains critical information necessary for system operation. The failure of this system during a critical mission might result in the collapse of combat coordination.

4.2 Computer Functional Requirements

Although the functional requirements for the applications listed above vary, there is a significant degree of commonality in basic throughput and storage needs. Within the Navy, there is a growing tendency to employ high level languages for design implementation. In order to facilitate the use of these languages, the simplex computer should comprise a 16-bit processor with 16K to 1M words of memory. Furthermore, the hardware configuration must support automatic fault detection, diagnosis and repair.

Since the NSFTC creates a multiprocessor, multiprogramming environment, an operating system must be selected that is capable of supporting complex program interactions. As indicated earlier, the most critical of these interactions is system synchronization prior to voting; system synchronization is also one of the most difficult interactions.

The requirements indicated above are more formally specified in Tables 4-1 through 4-4.

TABLE 4-1 Fault-Free System Operation
1. The computer system shall support multi-processing and multi-programming.
2. The computer system shall be capable of maintaining hot stand-by spare processors current with respect to data and function.
3. Sufficient memory shall be available to be compatible with the needs of high level programming languages.
4. Each simplex processor shall be capable of efficiently supporting the execution of high level languages.

TABLE 4-2 Fault Detection
1. Fault detection shall be performed in a timely manner to prevent propagation of fault-damaged information leading to permanent data corruption or erroneous outputs to peripheral devices.
2. Fault detection algorithms shall be transparent to the application software.
3. Fault detection algorithms shall detect all permanent and transient faults in the single board computers and redundant bus.
4. Fault detection algorithms shall detect all single permanent and transients faults in the Input/Output Controllers (IOC).
5. A fault log shall be kept and updated by the fault detection algorithms. This log shall be used to distinguish between permanent and transient faults.
6. In the event of single error correction in memory, the associated processor shall rewrite the faulty memory location with the corrected word.

TABLE 4-3 Fault Diagnosis
1. Fault diagnosis shall be capable of locating a fault the least replaceable unit. 2. A fault log shall be used to distinguish between permanent and transient faults. The classifier function is to be determined.

TABLE 4-4 System Repair
1. Automatic Repair a. The system shall dynamically determine the availability of spares. b. A failed system resource shall be isolated such that it cannot adversely affect the system. c. Resources not utilized in a hot stand-by mode shall be updated prior to use as a replacement for a failed component. d. Safe shut-down shall be used when degradations are no longer possible. 2. Manual Repair a. The system shall be provided with extensive off-line diagnostic software. b. The system shall be configured such that offline diagnosis does not interfere significantly with tasks still operational. c. All failed resources shall be configured to be removable from the working system for replacement.

5.0 Feasibility Analysis

The feasibility of developing the proposed fault tolerant computer system depends on demonstrating the reliability and availability increase of that system over simplex systems with a corresponding reduction in life-cycle cost. The following subsections discuss reliability, or the probability that a system will work correctly at a given time; external and internal sparing, two approaches to ensuring operational readiness; availability, which is the percentage of time a computer system is running; and maintainability, or the cost of system maintenance.

The analysis performed in this section will assume perfect coverage to simplify computations. Coverage is the conditional probability that a fault-tolerant system will recover from a fault given that a fault occurs. System reliability is a geometric series in coverage. Coverage is typically 0.99 or greater for a well-designed system.

5.1 Reliability Considerations

Reliability at time T is defined as the probability of a system working correctly at time T, given that it was working at time 0. It is a function of the aggregate failure rate of the components in the system and the capability of self-repair mechanisms in fault-tolerant computer architectures. The quantity of self-repair hardware must be balanced against the cost of that hardware. Therefore, analysis is necessary to determine where re-

dundancy should be added and how much redundancy is needed.

The starting point for any such analysis is to specify an application such that results are most meaningful. The applications assumed will be low, medium and high performance control computers. These computers will be assumed to require 0.98 probability of survival for a period of two years. Each system will be assumed to be constructed from large scale integration (LSI) components. It is usual to assume that the reliability of a semiconductor component at time t to be $e^{-\lambda t}$ where λ is the failure rate. In a simplex configuration, overall system reliability is computed from the series reliability of the components.

Failure rate is generally difficult to determine because it is a function of many operational parameters. However, failure rate may be estimated by extrapolation of currently accepted estimates. It is usual to assume a failure rate in the range of 10^{-7} to 10^{-6} failures per hour for LSI chips of complexity 10,000 gate equivalents [ARRO80] This is equivalent to a failure rate of 10^{-11} to 10^{-10} failures/hour per gate. In metal-oxide-semiconductor (MOS) devices, a gate is approximately four transistors. Static memory requires six transistors. Due to the regularity of design in a memory cell, it will be assumed that, on the average, a cell has a failure rate 1.0 to 2.0 times that of a gate. In any case, this corresponds to a failure rate on

the order of 10^{-11} to 10^{-10} failures/hour per memory cell. Well-designed power supplies are expected to have reliability on the order of 10^{-6} failures per hour.

Table 5-1 summarizes the three simplex systems discussed above and indicates optimistic subsystem and aggregate failure rates for each, based on extrapolation of the above gate and memory cell failure rate data. The increased failure rates for the central processors and memory for increasing performance level is due to the use of greater numbers of smaller, higher speed integrated circuits.

Assuming the values in Table 5-1, the probability of each system surviving for two years without any fault is summarized as:

Low performance system: 0.93
Medium performance system: 0.76
High performance system: 0.38

These numbers indicate that the simplex computer systems alone cannot achieve the required 0.98 reliability for the two year period. Meeting this requirement necessitates redundant components. The next issue to be explored is some of the options available for inclusion of this redundancy.

TABLE 5-1
Conservative Computer Failure Rate

SUBSYSTEM	LOW PERFORMANCE	MEDIUM PERFORMANCE	HIGH PERFORMANCE	
Memory	(4K x 8)	(16K x 16)	(64K x 16)	Size
	1.6	12.8	51.2	Rate
Processor	(10K gates)	(20K gates)	(50K gates)	Size
	0.5	1.0	2.5	Rate
I/O	1.0	1.0	1.0	Rate
Power Supply	1.0	1.0	1.0	Rate
Aggregate	4.1	15.8	55.7	Rate

Failure rate x 10^{-6} failures/hour

5.1.1 External Sparing

The current Naval approach to maintaining a given level of operational readiness is to provide duplicate systems that are used to replace failed systems. External sparing is the term generally applied to this philosophy, in which the computer system itself contains no redundancy. Failure recovery is thus achieved either by automatic or manual disconnection of the entire failed computer and connection of a spare. The analysis assumes perfect coverage and identical failure rates for unpowered spare and

powered active modules. This latter assumption has been determined to be valid when applied to MOS LSI because this technology is characterized by very low junction temperatures. As a consequence failure mechanisms accelerated or initiated by high junction temperatures have no effect. This study uses a standard reliability prediction model is used that was reported in the fundamental paper [BOCS69]. For q active modules, s spares, perfect coverage and identical powered and unpowered failure rates, reliability is computed from the equation:

$$ {}_cR_s^q = R^q \sum_{k=0}^{s} \binom{k-1+q}{k} c^k (1-e^{-\lambda t})^k $$

where λ is the failure rate , R is the basic module reliability and c is the coverage which is equal to one by assumption.

Figure 5-1 plots the reliability of an externally redundant system at two years, based on a system failure rate of $4.1*10^{-6}$ as a function of the number of active modules and spares. The two-year reliability is plotted on the X-axis, and the number of spares is plotted on the Y-axis. Solid lines in the figure represent reliability curves for different configurations of active processors. For example, one application may require only a single active processor, while another may need two or more. Dashed lines indicate the level of replication utilized. For example, the dashed line labeled X2 intersects the solid reliability curves where twice as many spares as active modules are

needed. Figure 5-2 is similar to Figure 5-1, but the failure rate is set to $55.7*10^{-6}$ failures/hour.

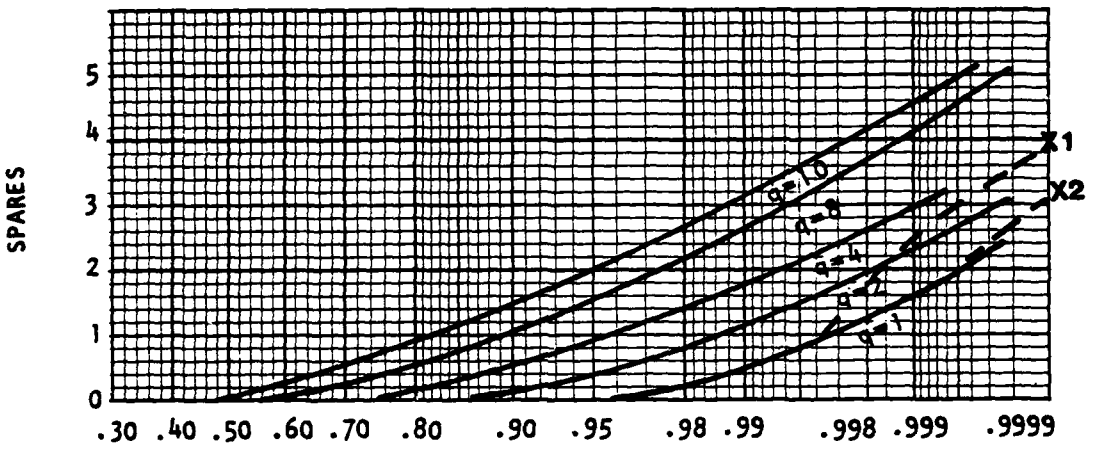


FIGURE 5-1 2 YEAR RELIABILITY ($\lambda = 4.1 \cdot 10^{-6}$ FAILURES/HOUR)

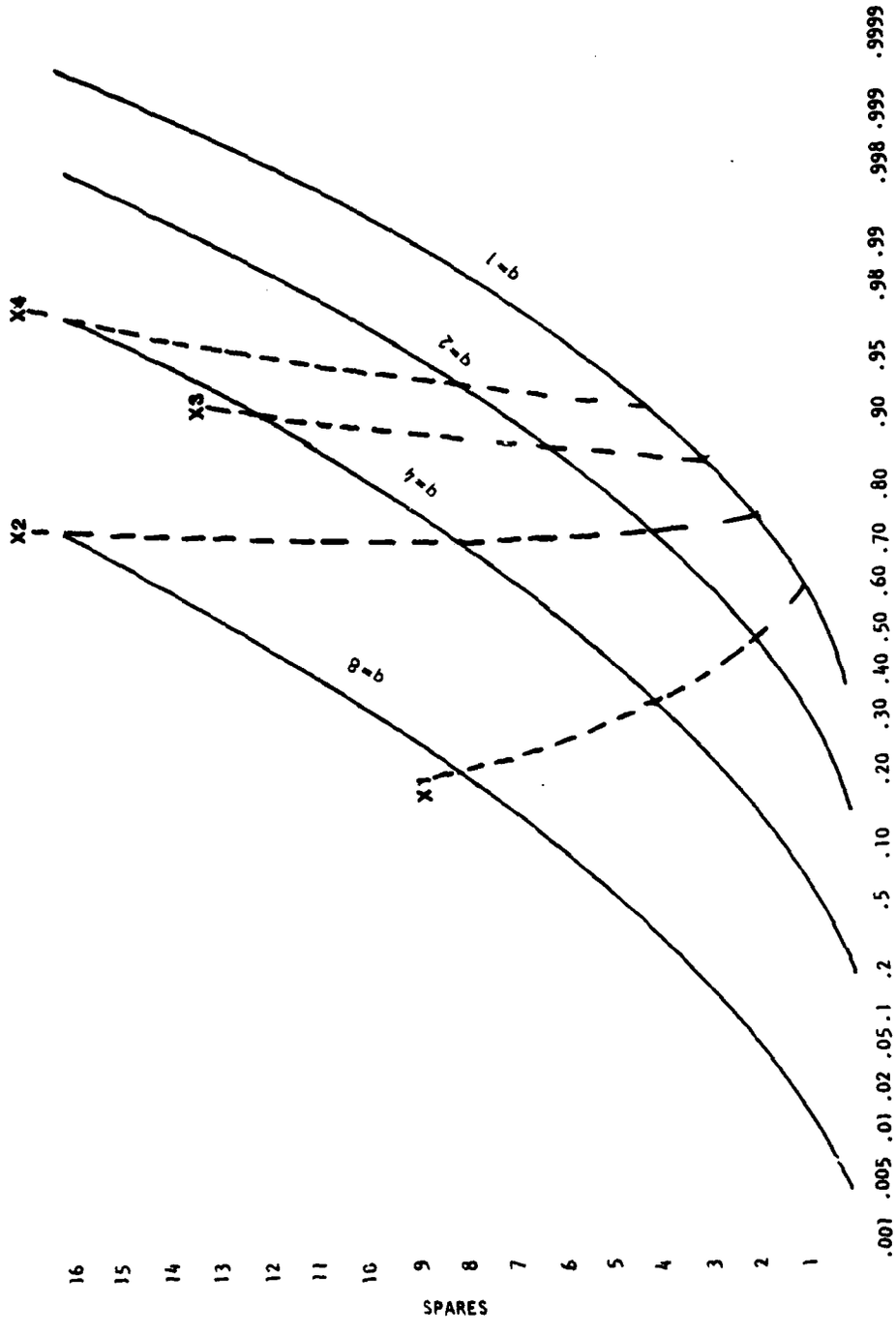


FIGURE 5-2 2 YEAR RELIABILITY ($\lambda = 55.7 \cdot 10^{-6}$ FAILURES/HOUR)

From these figures the following conclusion may be drawn:

> The cost of adding redundancy increases when the component reliability decreases. To achieve a practical system, reliability of the individual replaceable modules should be in excess of 0.8 at two years. Otherwise the cost of sparing quickly becomes untenable. It is therefore attractive to make use of internal redundancy to improve the reliability of the individual modules.

5.1.2 Internal Sparing

It was previously noted that the three theoretical computer configurations cannot achieve the required reliability without some form of redundancy. The use of external sparing will be practicable only for the low performance computer since it is the only system with a two-year reliability greater than 0.8. The other two configurations cannot be used unless either a large number of spares are provided or there is an improvement made in the basic component reliability.

An examination of Table 5-1 indicates that memory is the least reliable component in each system. It is therefore reasonable to bolster memory reliability by adding protection circuitry. The next least reliable component is the high performance processor. It is instructive to compute the reliability of the three postulated systems first assuming perfect memory, then with perfect memory and processor, and finally with only an unreliable power supply. These estimates are tabulated in Table 5-2.

TABLE 5-2 Two-Year Reliability			
	LOW	MEDIUM	HIGH
Non- redundant computer	.93	.76	.38
Perfect memory	.96	.95	.94
Perfect memory, processor	.97	.97	.97
Faulty power only	.98	.98	.98

The results indicated in Table 5-2 point to the benefit of internal redundancy, especially in the medium and high performance systems. Significant reliability improvements are obtained by designing fault-tolerant computer systems. Results obtained for externally redundant schemes indicate that providing low-reliability spares for low-reliability active computers yields some reliability improvement. In some cases, however, the number of spares required for a given level of reliability is impractically large.

Another conclusion to be drawn is that additional system reliability improvements necessitate improving power supply reliability. In a fault-tolerant computer application, such improvement may be derived from multiple, independent power supplies configured in a masking arrangement.

Transient soft memory errors generally occur at a rate one or two orders of magnitude more often than hard failures. From this standpoint, memory reliability will be greatly improved by providing a mechanism internal to the memory subsystem to correct soft errors. Bolstering memory system reliability is usually implemented via error correcting/detecting codes. For easy implementation and good error-handling, Hamming codes are usually selected to protect memory. It is known that a Hamming single error correcting code for an b-bit word will require p parity bits satisfying the inequality:

$$2^P \geq b + p + 1$$

This code results in a Hamming distance of three; double error detection requires a Hamming distance of four. Hamming distance is defined as the smallest number of bits that must change in a valid code-word to transform that word into another valid code-word. Implementing double error detection thus requires addition of one more parity bit than is necessary for single error correction.

5.2 Availability

Availability is defined as the percentage of time a computer system is running compared with the total time. The significant parameters in determining availability are mean-time-between-failure (MTBF) and mean-time-to-repair (MTTR). Expressed in terms of these parameters, availability is MTBF/(MTBF + MTTR).

MTBF is computed as the integral over all time of the reliability. For a simplex system, MTBF is 1/(failure rate). From Table 5-1, MTBF for the low, medium and high performance systems is 243,902, 63,291 and 17,953 hours respectively. For perfect coverage, it is reasonable to assume that the effect of fault tolerance is to lower the failure rate. A reliability of 0.98 in two years is equivalent to a failure rate of 1.15×10^{-6} which results in an MTBF of 867,210 hours.

MTTR is a more difficult number to estimate. Mathematically, it is equal to 1/(repair rate). In the fault-tolerant computer, repair will be performed automatically and quickly. It is possible, in fact, for a fault-tolerant design to mask the effects of a fault completely, which results in a MTTR of 0. On the other hand, repair of non-fault-tolerant computers requires the presence of skilled service personnel and needed spare parts. If these personnel and parts are available, an optimistic rate of two hours per repair may be assumed. In remote or dangerous applications, MTTR could be as long as several weeks.

Availability improvements with fault-tolerant designs are not significant in the postulated three system scenarios for MTTR less than 2% of MTBF. In general, MTTR will be much greater than 2% of MTBF. As MTTR approaches 2% of MTBF, availability drops to 98%. In many applications, this availability is unacceptable. From the results cited previously, 2% of the MTBF for the non-

redundant low, medium and high performance computers corresponds to 29, 7.5, and 2 weeks respectively. In the case of the fault tolerant-computer system, 2% of the MTBF is 103 weeks, a week short of the entire postulated two year mission.

Based on the results above, a fault-tolerant design can achieve significant improvement in availability when manual repair is not possible for extended periods of time. This implies that from an availability point of view, fault tolerance is most important in computer systems located in remote or dangerous locations. In general, however, fault tolerance greatly improves the availability of more favorably located systems. This is because real system failure rates are much higher than the optimistic values determined in the discussion of reliability considerations. Furthermore, MTTR in real systems must include preventive maintenance time in addition to corrective maintenance time. Availability of non-redundant real systems may be as low as 30% for highly complex array processors and range up to 90% or more for simple systems. Fault tolerance can easily yield availabilities in excess of 98%.

5.3 Maintainability

The life-cycle cost of a computer system is composed of the costs associated with preventive and corrective maintenance. The rates at which each form of maintenance is applied is strongly connected with the reliability of the system. Highly reliable

systems require corrective maintenance much less often than unreliable systems. Furthermore, preventive maintenance may be scheduled less often for more reliable systems.

There are two maintenance policies that are distinguished by the manner in which preventive maintenance is scheduled. In one policy, preventive maintenance is rescheduled after a corrective maintenance action. This policy makes the assumption that the ages of the components within the system are noted at the time of corrective maintenance, and the replacement period T therefore starts at that time. When corrective maintenance occurs too often the period T must be shorted. In the second policy, the preventive maintenance schedule remains fixed whether or not corrective actions have been taken. This policy is generally useful when it is not practicable to keep a record of corrective maintenance or when a preventive maintenance schedule must be fixed for reasons of logistics or maintenance contract.

Within the realm of the first maintenance policy it can be shown [LAMA80] that the cost per operating hour C_T is equal to:

$$C_T = C_C(1-R(T))/\int_0^T R(t)dt \; + \; C_P(R(T))/\int_0^T R(t)dt$$

where C_C and C_P are the cost of corrective and preventive maintenance procedures respectively. Using the assumption of a fixed failure rate system in which reliability is equal to $e^{-\lambda t}$, this equation simplifies to:

$$C_T = C_C \lambda + C_P(e^{-\lambda T}) \lambda / (1-e^{-\lambda T})$$

The cost C_T and reliability R(T) is tabulated in Tables 5-3 through 5-6 for varying T and failure rates corresponding to the three simplex computer systems and a fault-tolerant computer system. Cost per preventive maintenance is assumed to be $20.00 and the cost per corrective maintenance $100.00.

Examination of these tables indicates that maintenance cost monotonically decreases with increasing T. This is the result of the constant failure rate assumption. If a Wiebull failure distribution is assumed, for example, there is a knee in the cost versus T curve such that T may be optimized as a function of cost.

Another effect of varying T is the resultant reliability. The tables indicate the expected decrease in reliability with T. Selecting T such that reliability remains greater than 0.98 requires T = 4,500 hours, 1,000 hours, 350 hours and 20,000 hours respectively for the low, medium, high and fault-tolerant configurations. Due to the monotonically decreasing maintenance cost function, the smaller values of T correspond to large values of cost. The maintenance costs computed for reliability > 0.98 are $.0048, $.021, $.06 and .001 for the four systems. From these numbers, maintenance costs of the fault-tolerant computer are reduced by factors of 4.8, 21 and 60 from the three nonredundant

systems.

TABLE 5-3
Failure rate = 10^{-6} failures/hour

T(HOURS)	RELIABILITY	COST
10000	.990050	.0020
11000	.989060	.0019
12000	.988072	.0017
13000	.987084	.0016
14000	.986098	.0015
15000	.985112	.0014
16000	.984127	.0013
17000	.983144	.0012
18000	.982161	.0012
19000	.981179	.0011
20000	.980199	.0010
21000	.979219	.0010

TABLE 5-4
Failure rate = $4.1*10^{-6}$ failures/hour

T(HOURS)	RELIABILITY	COST
500	.997952	.0404
1000	.995908	.0204
1500	.993869	.0137
2000	.991834	.0104
2500	.989802	.0084
3000	.987776	.0070
3500	.985753	.0061
4000	.983734	.0054
4500	.981719	.0048
5000	.979709	.0044

TABLE 5-5 Failure rate = $15.8*10^{-6}$ failures/hour		
T(HOURS)	RELIABILITY	COST
500	.992131	.0414
1000	.984323	.0214
1500	.976579	.0145

TABLE 5-6 Failure rate = $55.7*10^{-6}$ failures/hour		
T(HOURS)	RELIABILITY	COST
50	.997219	.4050
150	.991680	.1383
250	.986172	.0850
350	.980694	.0622
450	.975247	.0495

Considering the second policy, the cost per operating hour is computed as:

$$C_T = C_C \lambda + C_P \lambda_{pm}$$

where λ_{pm} is the preventive maintenance rate. Tables 5-7 through 5-10 tabulate this cost for various failure rates and preventive maintenance rates.

The results in the Tables again indicate a monotonically decreasing cost function with respect to time. As with the first maintenance policy, the cause of the monotonic decrease is the constant failure rate assumption. The savings in maintenance cost is not significant for this second policy, however aggregate costs for all systems are much higher than for the first maintenance policy. The primary reason for this result is that preventive maintenance is performed much more frequently in this policy than the first policy. This implies that the system is being operated in a fault avoidance mode, that is, maintenance is performed often enough to make system failure highly unlikely. Such procedures are very expensive for all computer configurations as indicated by the two orders of magnitude increase in cost.

TABLE 5-7

Failure rate = $55.7*10^{-6}$ failures/hour	
PM INTERVAL (HOURS)	COST
100	.2056
200	.1056
300	.0722
400	.0556
500	.0456

TABLE 5-8
Failure rate = $15.8*10^{-6}$ failures/hour

PM INTERVAL (HOURS)	COST
100	.2016
200	.1016
300	.0682
400	.0516
500	.0416

TABLE 5-9
Failure rate = $4.1*10^{-6}$ failures/hour

PM INTERVAL (HOURS)	COST
100	.2004
200	.1004
300	.0671
400	.0504
500	.0404

TABLE 5-10 Failure rate = $1.0*10^{-6}$ failures/hour	
PM INTERVAL (HOURS)	COST
100	.2001
200	.1001
300	.0668
400	.0501
500	.0401

5.4 Summary of Feasibility Study Results

In this analysis, the feasibility of fault-tolerant computer design was based on computing reliability, availability and maintenance cost improvements for three computer systems. In all three parameters, fault-tolerance was found to be of significant value. Furthermore, the results of the reliability analysis indicate that fault tolerance should be implemented via internal rather than external redundancy.

The most significant result of this study is the cost of

maintenance for the three non-redundant systems as compared with a fault-tolerant system. The results of this analysis indicate a clear advantage in the use of a fault-tolerant design. That analysis, however, predicted costs based only on the average cost of a service call. The actual costs for these calls should include the loss of the entire system during the service period. In most of the Navy's embedded computer systems, the actual cost for maintenance is two or more orders of magnitude greater than the cost estimates used above. Based on the lower cost maintenance policy 1, a 0.98 or greater reliability for a two year mission and a preventive and corrective maintenance cost of $2,000, and $10,000 per call, respectively, ownership will cost $.48, $2.10 and $6.00 for the low, medium and high systems. The fault tolerant system, however, will cost only $.10 for the same period of time. The savings in maintenance cost per application of the fault tolerant computer multiplied by thousands of systems in use can save tens of millions of dollars per year.

The fault-tolerant computer architecture recommended by this research provides internal redundancy for the most likely sources of failure as determined in this section. Furthermore, the architecture reflects the need to provide flexibility for tailoring performance, reliability, availability and maintainability for specific applications. The use of LSI-based off-the-shelf components to implement the fault tolerance features of the proposed computer yields a design that is far superior to any

simplex computer of equivalent performance, while not affecting the initial purchase price significantly.

6.0 NSFTC THEORY OF OPERATION

The NSFTC is protected against physical fault-induced failures via a hierarchical fault detection mechanism and a simple recovery technique. Hierarchical fault detection implies that check mechanisms exist at several levels within the architecture. Higher level checking catches faults missed at lower levels. This method greatly increases the probability of fault detection while not significantly adding to the quantity of redundant components required.

The purpose of this section is to detail the fault-tolerant design concepts used in the NSFTC as a prelude to the hardware and software discussions in Sections 7 and 8 respectively. These concepts include a fault detection hierarchy, self-checking circuits, synchronization considerations, naming conventions and fault recovery procedures. A more general discussion of fault-tolerant computer design concepts is found in Appendix I.

6.1 Fault Detection Hierarchy

Hierarchical fault detection is commonly used in fault-tolerant computer system design. As indicated above, a hierarchy is used both to increase fault coverage and to minimize the quantity of redundant circuitry. Lower-level checking is performed by small, special purpose hardware. Higher-level checking involves more general purpose hardware executing special purpose software.

Fault detection is the cornerstone of fault repair. It is the fault detection algorithms that call fault recovery routines once a fault is detected. As a result, it is essential that these routines detect a very high percentage of faults with a correspondingly low false alarm rate.

In the NSFTC, the lowest level fault detection hardware consists of Hamming encoding/decoding circuitry in the SBP memory and self-checking circuit design in the IOCs. These checks must be done at the hardware level for reasons of fault coverage and throughput. The IOCs are capable of detecting in excess of 99% of their own faults and are capable of taking appropriate corrective action. Except for memory-protecting codes and multiple internal bus connection hardware, the SBPs contain no other form of redundancy.

Faults in the SBPs are detected at the next level. In this level, error-detecting codes protect internal bus transfers and voting checks the operation of the SBPs. Many massive faults not detected by the lowest level will result in the inability to correctly format and participate in internal data transfers. Should a faulty SBP somehow still be capable of participating in internal data transfers, erroneous information will be detected via voting.

Voting is performed in the self-checking IOC, which also generates synchronization signals for the system. After a specified

period of time, all SBPs should be synchronized and be outputting critical information for examination by the IOC. Should an SBP not respond to the synchronization request within the allotted time period, the IOC will assume that it is faulty. Upon receiving two or more data transfers from the SBPs, the IOC votes and determines whether or not a disagreement has occurred. Voting will detect any number of faults in a single SBP and is also capable of detecting multiple uncorrelated faults. The voting process is performed in a self-checking device so that there is never any question as to whether or not that device is behaving as expected.

6.2 Self-Checking Check Circuits

A check circuit is a redundant system element used to verify the correct operation of another part of the system. The concept of a check circuit is illustrated in Figure 6-1 which shows two circuits A and B. Circuit A is a functional element that, when fault free, produces outputs that are components in the overall computation execution. These outputs are redundantly encoded such that circuit B is capable of discerning the difference between good and bad outputs. Thus the outputs of circuit A are grouped into two sets: G - good and B - bad. Check circuit B implements a mapping from G into g and B into b when no check circuit faults are present. The difficulty, however, is that check circuits are usually implemented with semi-passive signal

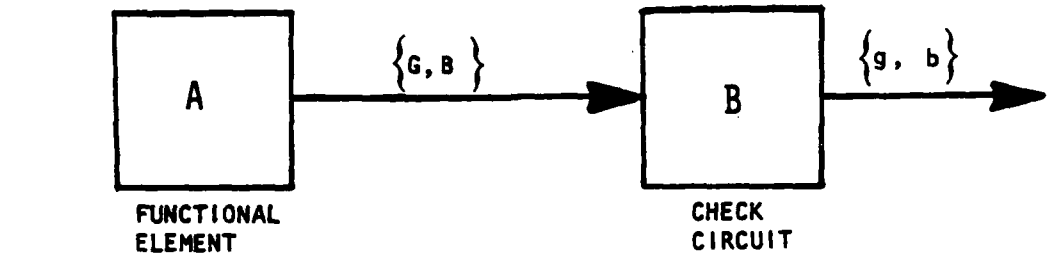
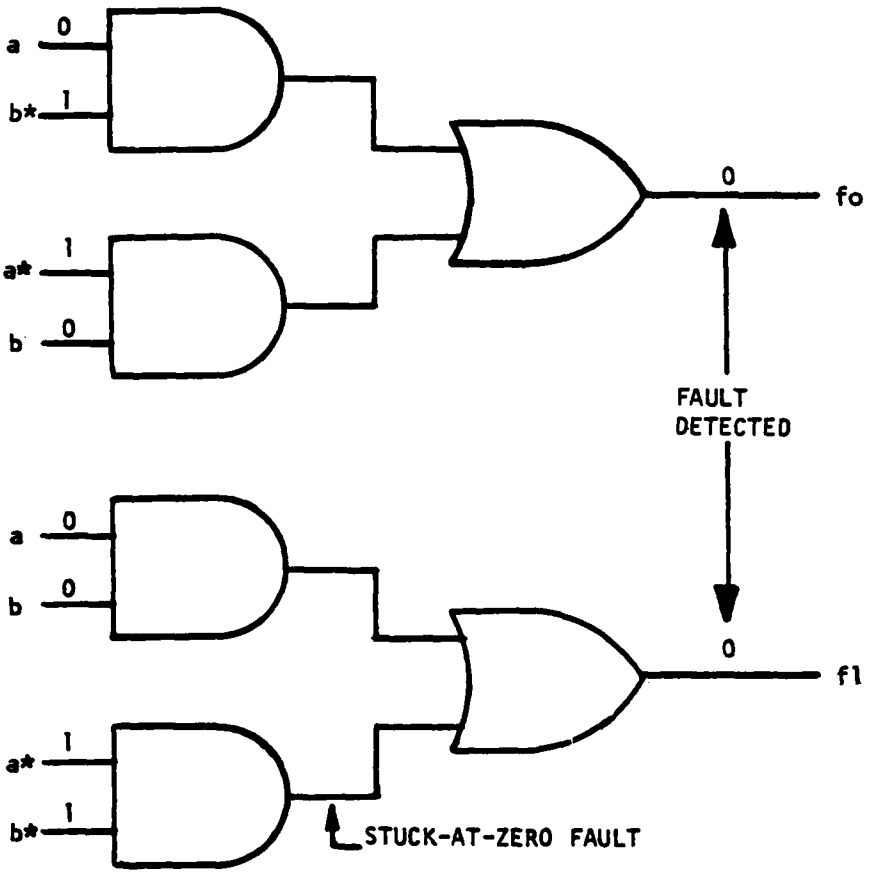


FIGURE 6-1 CLASSICAL CHECKING CONFIGURATION



FAULT-FREE TRUTH TABLE

A		B			
a	a*	b	b*	fo	f1
1	0	1	0	0	1
1	0	0	1	1	0
0	1	1	0	1	0
0	1	0	1	0	1

FIGURE 6-2 SELF-CHECKING AND

wires. Thus, there is no way to test whether or not the check circuit will correctly flag the presence of a fault when one is detected in the output of circuit A.

Signal wires may be classified as being either active or semi-passive. Active signal wires are continuously changing value during normal operation, whereas semi-passive wires change value only when exceptional conditions occur, for example, the detection of a malfunction. When semi-passive signal wires are used in the implementation of check circuits, it is not possible to simultaneously sensitize all paths through the check circuit to test for the presence of internal circuit faults while maintaining a fixed output value. A *means* of sensitizing paths through the check circuit without affecting the output value is to replace the semi-passive signals wires in the implementation of the check circuit with redundant active signal wires. Since these redundant wires continually change value, check circuit path sensitization and testing may be done concurrently with normal operation.

In a fundamental paper, Carter et. al. [CAWJ71] have reported on a mapping from conventional single-line logic variables to redundant pairs of wires. The mapping is defined by:

$$\begin{aligned} M:\ & [(1,0),\ (0,1)] \longrightarrow 1 \\ & [(0,0),\ (1,1)] \longrightarrow 0 \end{aligned}$$

In this redundant representation, the conventional logic

values have two representations. Each wire in a redundant pair may assume both conventional logic values of 1 and 0 without affecting the equivalent single line logic value. In fact, both lines are designed to change value frequently to allow complete testing of a circuit. A morphism exist between conventional Boolean algebra and a Boolean algebra over the line pairs. A morphism is defined as a mapping from one algebra into another in which order is preserved.

Figure 6-2 shows a circuit that implements a self-checking AND function over two redundant signal pairs A = (a, a^*) and B = (b, b^*). As in a conventional two input AND, the application of TRUE at both inputs results in a TRUE output. In the AND gate over the redundant pairs, this is still the case. This fact forms the basis for the use of this circuit in self-checking check functions.

Figure 6-3 shows two identical circuits C and C'. The outputs of C' are complemented and input to the redundant AND circuit along with outputs from circuit C. When no faults are present, the signals applied to the AND gate will always be complementary, or equivalent to the single-line logic value of 1. Therefore, the output of the AND will be a redundant 1 when no faults are present. Should circuit C disagree with C', then one of the inputs to the AND gate will be a redundant 0, which results in an

output of redundant 0. Thus the AND is capable of acting as a check circuit.

It remains to be seen that the redundant AND is capable of detecting its own internal faults. Assume that the wire indicated by an X in Figure 6-2 is broken as the result of a fault. For the input pattern indicated in the figure, the resulting output is a redundant 0, and thus the fault is detected. A careful analysis of this redundant AND indicates that for all stuck-at and bridging faults, there exists at least one valid input combination (both input pairs equivalent to 1) that will produce an invalid output (output equivalent to 0) [SIAV81].

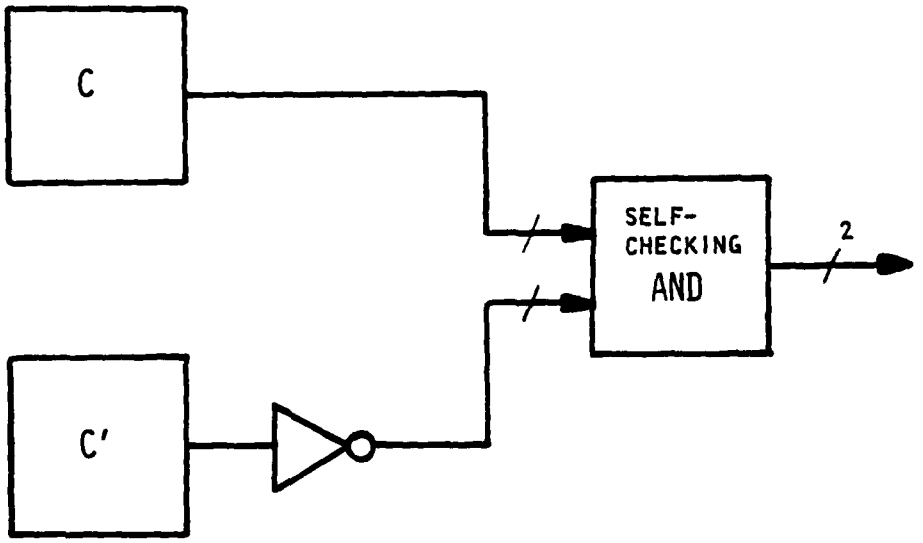


FIGURE 6-3 SELF-CHECKING CHECKER

It is not possible to distinguish between a faulty check circuit and an invalid input. As a result, it is necessary to consider the checker as part of the circuits being checked. Any fault indication at the check circuit output necessitates replacement of the entire unit.

The example of Figure 6-3 indicates that the circuits under test must be duplicated. As a consequence, it is not practical to use self-checking circuits everywhere in the system design. The NSFTC requires only a self-checking IOC. As will be seen in Section 7, this is a relatively small circuit constructed from off-the-shelf LSI components and therefore does not add significantly to the hardware cost.

6.3 Synchronization

In a loosely coupled voting system it is necessary to periodically synchronize processors in preparation for voting. As processors drift farther away from synchronism, the time required for voting increases. Consequently, processors should, as closely as possible, be executing the same instruction at the same time. Normal differences in the internal clock period of each processor are the primary component in system entropy.

From a theoretical point of view, the synchronization problem necessitates a minimum of four independent clocks. Practical experience, however, indicates that for all practical fault con-

siderations only three clocks are required. The unique architecture of the NSFTC will necessitate only two clocks.

The scenario envisioned makes use of the IOC as the primary synchronizing component. On cold-start, one IOC will be designated as the source for system synchronizing pulses. Another IOC will be designated as a monitor of the first. The self-checking feature of the IOC ensures that the monitoring unit is capable of correctly executing the monitor software. Three cases may arise as a result of an IOC failure.

6.3.1 Case 1: Failure of the Initial IOC

The cessation of system-wide synchronization pulses indicates the loss of the IOC charged with that task. The monitoring IOC will detect the loss of these pulses via a simple watchdog timer routine. That IOC will take over the job of synchronization and request an assignment of another monitor IOC should there be an additional spare. The course to be followed in the event no spare IOCs are present depends on the application.

6.3.2 Case 2: Failure of the Monitor IOC

The failure of the monitor IOC could lead to system failure unless simple precautions are taken. Depending upon the application, loss of the monitor IOC will be detected either by the SBPs that use this IOC, by IOC-IOC status checking, or both. In the event of such a failure, the master IOC will attempt to select another synchronization monitor should a spare IOC be available.

In the event that another IOC is not present, the application will determine whether to shut down or proceed.

6.3.3 Case 3: Disagreement

This third case is the most difficult to deal with. It is possible for a clock to fail within an IOC such that it either runs much faster or much slower than the other IOC. In this situation, the monitor IOC will find a disagreement with the master IOC. When such a condition occurs it is not possible for the two IOCs to resolve the inconsistency without external assistance. In multiple IOC configurations, the monitor IOC could request independent verification from another IOC. Depending on the outcome, either the master or the monitor IOC will be removed. When additional IOCs are not present, the application determines the next step. In applications requiring maximum availability, the disagreement can be resolved by involving the SBPs. These can be requested to monitor the synchronization pulse timing with respect to their own internal clocks, and information can be exchanged to determine which IOC is bad. In applications requiring maximum safety, an orderly shut-down will be initiated.

6.4 Naming Conventions

Each resource has a hard-wired hard name which is used in system initialization to establish an operational configuration. Each resource also responds to a variable soft name. During normal

operation, soft names are used for source and destination designations in data transfers and for establishing the identity of elements that make up a TMR group.

The dual naming convention is very important in a multi-processor, reconfigurable architecture. Hard names are equivalent to real resources whereas soft names refer to virtual resources. The use of virtual resources is a key component in providing transparency of the fault-tolerance features to the application software. The philosophy is to assign the task of resource binding to the operating system rather than the writer of the application.

6.5 Fault Recovery

Replacing failed resources with spares is readily accomplished through a hard and soft naming scheme as indicated above. Replacing failed components is accomplished by setting the soft name of a failed resource to null (such that it no longer responds to messages directed to that soft name) and then changing the soft name of a spare.

In the event that no spares are available, the application determines the recovery procedure. Applications that may continue without the failed resource will be allowed to proceed. Should the failed resource be an SBP, the system will be configured for comparison rather than voting. This is done to maxi-

mize reliability. In applications in which computer failure is potentially dangerous, the safest path to follow is a safe-shutdown. Refer to Section 7 for more discussion of recovery options.

Providing the ability to handle different recovery mechanisms will be the function of a Recovery Manager within the operating system. The Manager will keep track of the current configuration and be cognizant of the permitted recovery algorithms. It is presently envisioned that the Manager will be an implementation of a standard Markov modeling process for fault-tolerant computer systems. In this manner, the system configuration is described by a set of states. Each state represents the number of active and spare modules available. Furthermore, the model may be analyzed to predict, online, system reliability. This latter capability may be extremely useful in evaluating the results of delaying maintenance.

7.0 Hardware Design

The goal of this effort is to specify a flexible fault-tolerant computer configuration that can be constructed from off-the-shelf hardware and software as much as this is practicable. Flexibility is provided to meet both performance and reliability requirements. This approach permits tailoring the architecture for a specific set of requirements and results not only in a low cost design, but also in minimizing the risk involved in successful completion.

This section discusses the basic NSFTC system architecture illustrated in Figure 7-1. A redundant serial bus interconnects multiple SBPs to multiple IOCs. Data transferred over the bus will be encoded such that it is possible to detect faults in the transmission. This system may be configured with different numbers of resources for specific applications.

The key element in the architecture is the IOC. This device is responsible for executing the voting function and for providing a means to synchronize the SBPs prior to voting (refer to the previous discussion of voting). Due to the critical nature of this component, it is considered to be the system hardcore. It is internally self-checking (see the previous discussion) to detect its own faults. The IOC will be designed such that in the event of an irrecoverable internal fault, it will take itself offline. This action will prevent sending erroneous data to a

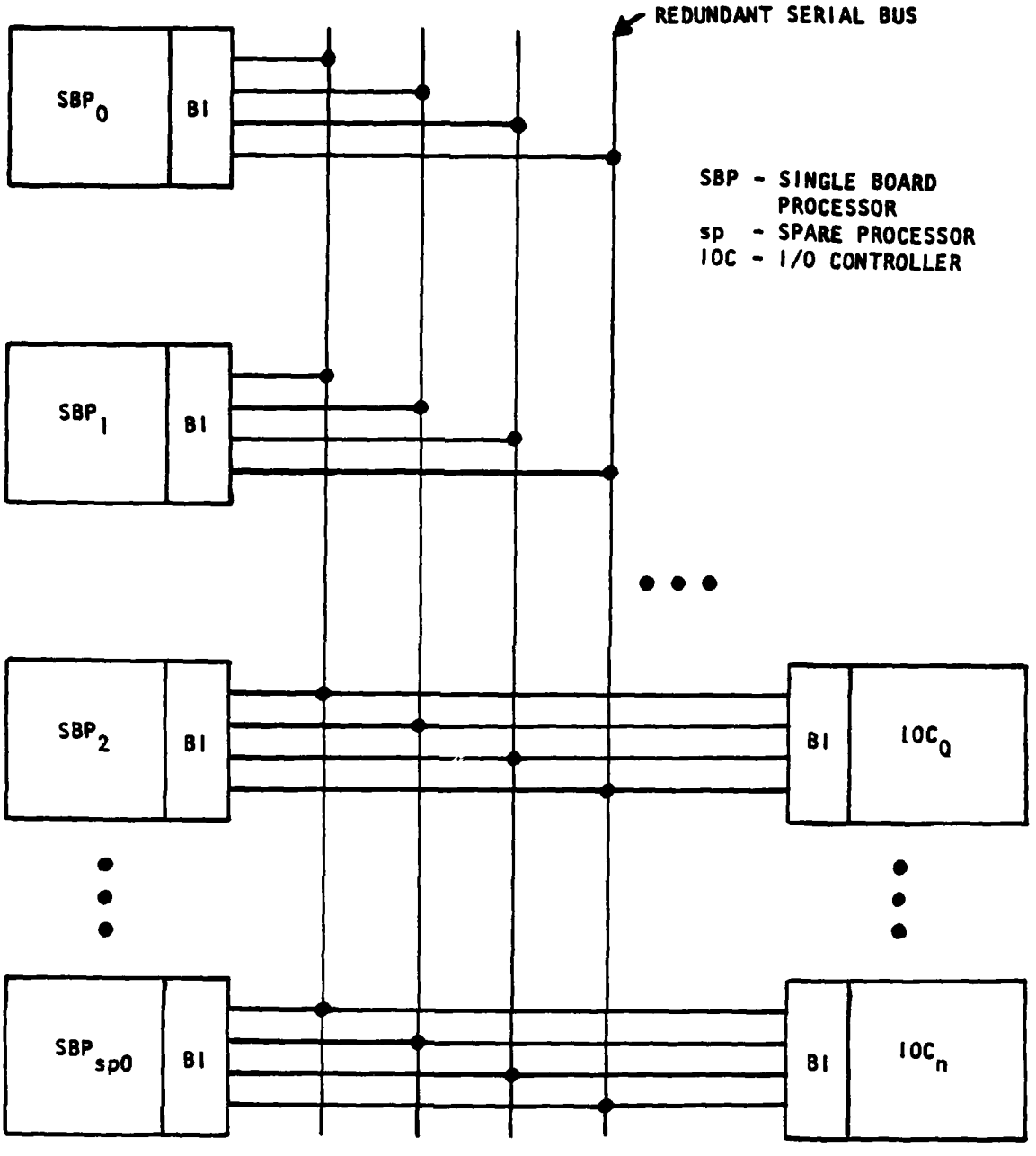


FIGURE 7-1 HIGH LEVEL ARCHITECTURE

peripheral. In order to continue operation in the event of an IOC failure, the system must be configured with at least two IOCs.

The redundant bus provides flexibility for assigning communication paths to the SBPs in order to maximize throughput and to allow system reconfiguration in the event of a bus failure. Error coding will be employed to enable fault detection on the bus.

A minimum-configuration system consisting of two SBPs and a single IOC is capable of detecting a fault and executing a safe shut-down. The reliability of this configuration is computed as the series probability that both SBPs are operational. This reliability will always be less than the reliability of a simplex computer as indicated in Figure 7-2. Thus this configuration should only be used when it is desired to prevent fault-damaged information from propagating to a peripheral device.

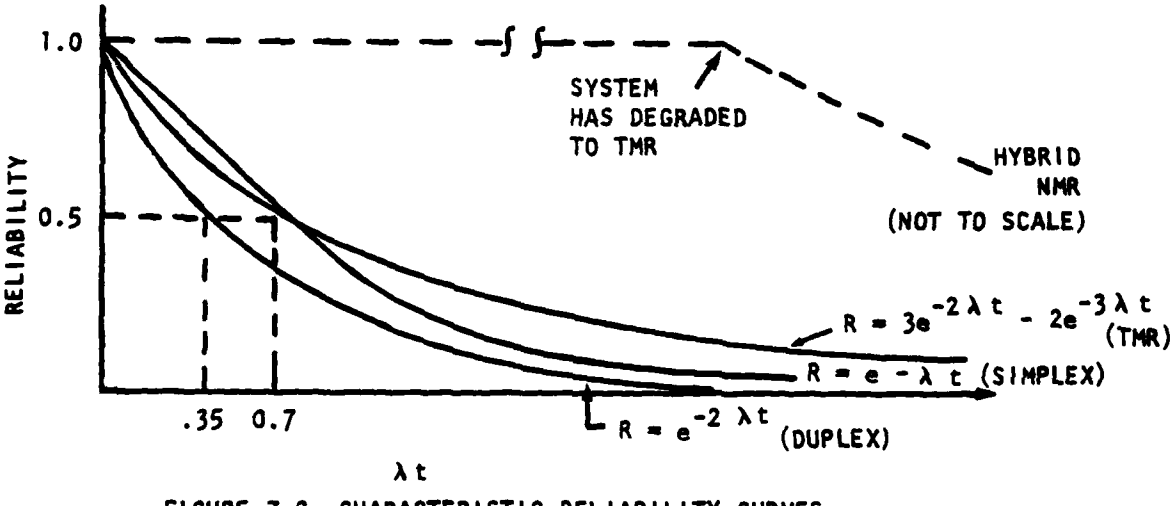


FIGURE 7-2 CHARACTERISTIC RELIABILITY CURVES

When three SBPs are available (TMR configuration), the system is capable of detecting and masking the effects of muliple uncorrelated faults. However, as Figure 7-2 indicates, the reliability of the system after the first failure becomes less than that of a simplex system. After the first failure, any second failure causes the immediate failure of the entire system. This is the same situation as in the duplex system described above. As a result, in the TMR configuration, the system should be degraded to a single working SBP if the application requires maximizing reliability. If safety is considered more important than reliability, the faulty SBP should be taken offline by the IOC and the outputs of the remaining working SBPs compared. Although this results in lower reliability, the IOC will be able to determine when there is another disagreement and order shutdown.

Finally, in a so-called hybrid n-modular-redundant (NMR) configuration, failed SBPs are put offline by an IOC and spare SBPs are enabled. This configuration is the most complex; however, as indicated in Figure 7-2 this results in the best reliability curves.

7.1 Single Board Processor (SBP)

Application software will be executed by the SBP within the system. These processors will be off-the-shelf components and, as such, will minimize hardware design risk. Furthermore, the NSFTC places almost no requirements on the redundancy features

on-board these modules. Consequently, these modules may be selected on the basis of their suitability to the application task. An error detection/correction memory is a recommended feature, but not essential. Soft memory errors are the most probable fault type to be encountered by the NSFTC. Consequently, providing local memory protection will repair a large number of faults without interrupting normal processing to run the recovery routines.

Figure 7-3 shows the architecture of an SBP. An internal bus connects the central processing unit (CPU) to read only memory (ROM), random access memory (RAM), local timing and the Bus Interface. The internal bus is not redundant and is selected to be compatible with interconnection requirements of the attached modules. Faults affecting this bus will result in communication or memory errors. Both of these types of errors will be detected during voting. The Bus Interface is composed of a Bus Controller and multiple Bus Adaptors that implement redundant connections to a redundant external bus. Other than isolation circuitry used in the Bus Adaptors to minimize external bus contamination in the event of a faulty Bus Interface, the Bus Interface is a simplex module. Bus Interface malfunctions will be detected either during external bus communication or via voting.

7.2 External Bus Architecture

The external bus provides a redundant communication path between

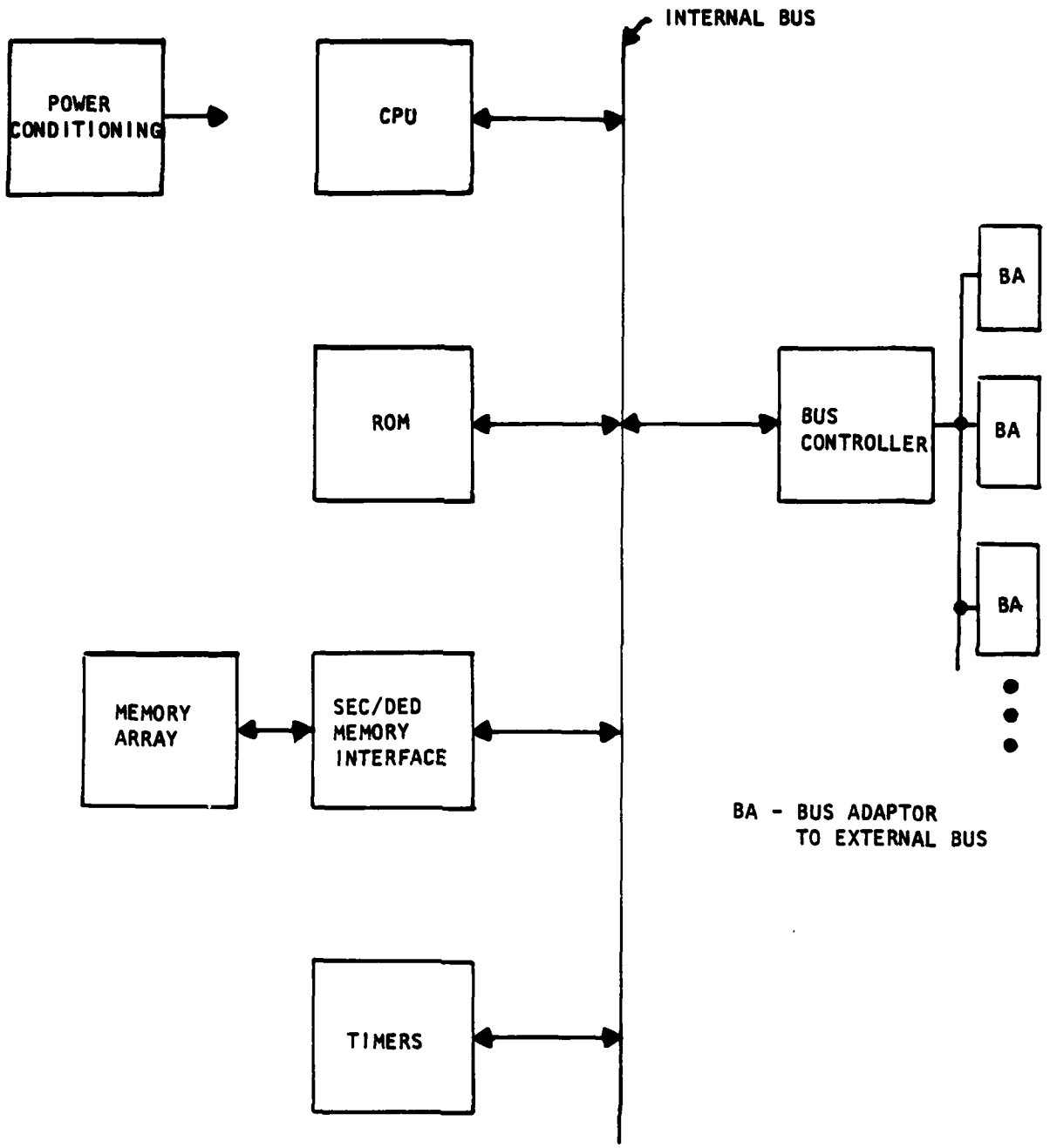


FIGURE 7-3 SINGLE BOARD PROCESSOR

the SBPs and the IOCs and is protected by error coding. The external bus consists of multiple serial data links and is organized such that the failure of any link does not cause the system to fail. Serial links are used to minimize the number of wires in the bus. In high performance applications, it may be necessary to increase the parallelism within this bus. For example, each communication path may be composed of two, three or more wires.

When all paths within the bus are operational, the SBPs may make use of the redundant bus to increase bandwidth. As permanent bus-related failures occur, SPBs will be required to share the bus more heavily, resulting in degraded performance. From the point of view of error detection, it is highly desirable for communication between SBPs and the IOC voter to occur over different paths. In this manner, any single bus-related failure results in corruption of information from one SBP which can then be voted out by the good results from the other two.

7.3 IOC

Figure 7-4 illustrates the architecture of the self-checking IOC. A redundant external bus interface, consisting of multiple Bus Adaptors and a Bus Controller, is identical to that used in the SBP. Unlike the SBP, however, the IOC is configured with two Bus Interfaces to provide the required duplication for self-checking. Dual single chip computers execute in clock step to manage

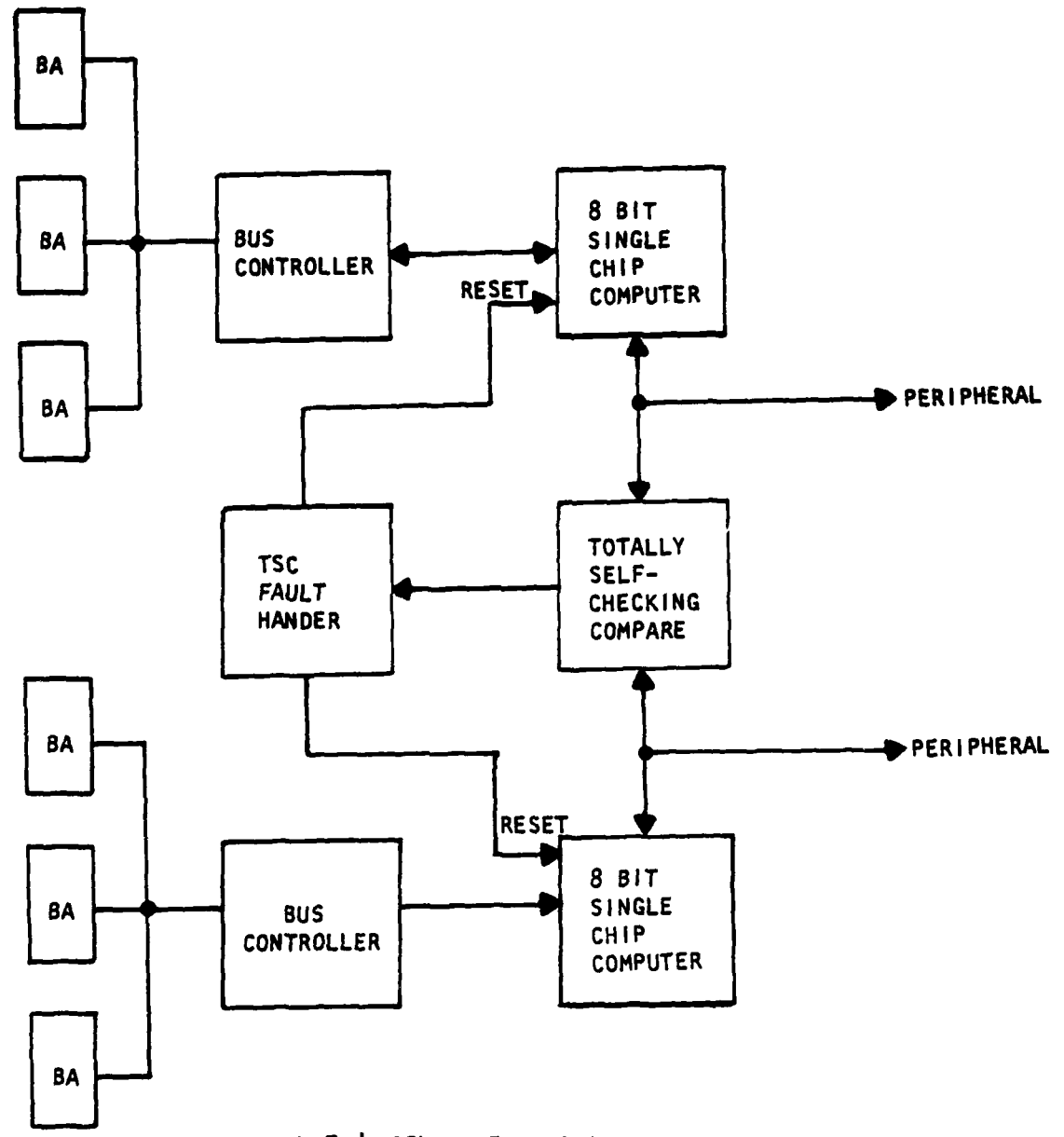


FIGURE 7-4 SELF-CHECKING IOC

input/output and voting operations. These computers connect to a totally self-checking (TSC) morphic AND Comparator. Any fault affecting either string through one of the Bus Interfaces or the single chip computer will result in a miscompare. In the event of a miscompare or a fault within the TSC Comparitor, the Fault Handler is activated. The Fault Handler is a small TSC finite-state automaton that issues reset and rollback commands to the IOC. The occurrence of any single fault is treated as a transient, and thus recovery is attempted by resetting each computer for several clock periods. Upon release of the reset, each processor executes a rollback procedure in an attempt to restart the failed input/output or voting operation. A second fault occurring during the recovery period causes the Fault Handler to permanently reset both processors, effectively removing the IOC from service. The IOC may also be removed from service via commands received on the external bus.

8.0 System Software

An operating system for the NSFTC must provide the usual services of resource sharing and task scheduling. In addition, the operating system must provide those features required for redundancy management. The added complexity of a multiprocessor system requires that the operating system be distributed to minimize the possibility of a single point failure.

The following paragraphs describe several considerations for an NSFTC operating system. Allocation of operating system features to specific NSFTC components requires a detailed analysis in order to minimize communication overhead. This study has focused on the feasibility and architectural issues of a fault-tolerant computer. Therefore detailed consideration of the operating system requirements is a major topic for future study.

There are three distinct operational modes that must be managed by the operating system. In normal operation, periodic communications must occur among resources to affect synchronization, voting and input/output. Within this operation, each SBP must schedule its own program execution to coincide with IOC operation and application task execution.

During the interval between the occurrence of a fault and the subsequent detection and recovery procedures (detection latency), the operating system must be sufficiently robust that it can tolerate the effects of the fault. This implies careful control

over shared databases and watchdog timers for input/output operations.

In the recovery latency period, the system has detected a fault and is in the process of isolating the faulty resource and implementing a repair action. The operating system must be tailorable to manage a variety of recovery mechanisms and configuration limitations.

The operating system is therefore composed of fixed and variable components. Fixed components will be external bus communication, task scheduling, input/output drivers, synchronization and voting software. Variability will be manifested in the recovery software. A structure supporting these features is indicated in Figure 8-1. This figure indicates the operating system core surrounded by the variable recovery mechanism. The implication of this arrangement is that the recovery mechanism is built from components within the core. Application software executes on the virtual machine defined by the recovery mechanism and core software. The application software executes on a logical machine that is defined by a memory size, processor and input/output mechanism. The logical machine contains no reference to the actual implementation of redundant components and fault detection/recovery capabilities. Software written for a non-redundant processor of the same type as used in the SBPs will execute without significant change in the NSFTC.

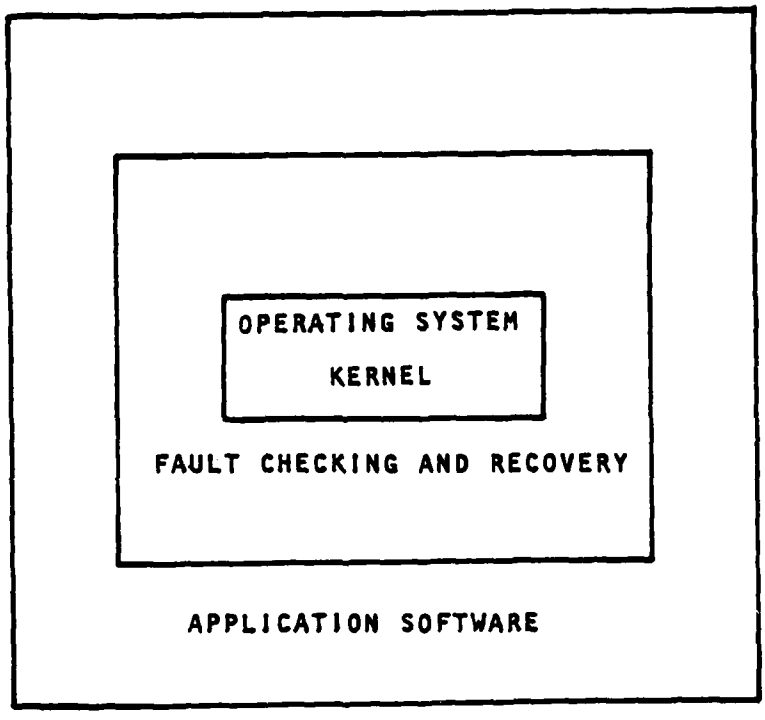


FIGURE 8-1 SOFTWARE HIERARCHY

The recovery mechanism may best be handled by implementing a Markov model of the state of the system. One such model described in [NGAV80] consists of a set of states, each indicating the number of spare and active modules remaining in the system at a given time. A state transition occurs in response to a fault. This model may be used both to guide the recovery strategy and, equally important, to make online estimates of the current system reliability. This latter use can be extremely important in determining whether or not the computer is capable of surviving a given length mission in its present state.

9.0 Reliability Analysis

The SBPs and IOCs are of the complexity of the medium and low performance computer systems, respectively. With an error correcting code, the memory subsystem for the SBP may be considered perfect, resulting in an aggregate failure rate of 3.0 failures/10^6 hours. Coverage for the NSFTC is expected to be in excess of 0.98. Tables 9-1 through 9-6 indicate the two- and five-year reliability for NSFTC configurations consisting of two, four and eight active modules for zero to nine spares and coverages equal to 0.95, 0.98 and 0.99. A configuration consisting of two active modules and one spare is analogous to a TMR system, for example. Reliability was computed using the model discussed in the feasibility analysis.

The tables indicate the strong influence of coverage on reliability. It is apparent that adding poorly covered spares does not have a great impact on reliability. This implies that redundancy should be used both to improve the reliability of the least replaceable modules and to increase coverage. Coverage can be increased by use of better fault detection and recovery mechanisms. In the NSFTC, fault detection will performed primarily by self-checking hardware and software comparison. These methods yield a good trade-off between hardware complexity and detection capability. Fault recovery is primarily a software function and as such may be designed to be as sophisticated as system performance and storage limitations allow.

An examination of Table 9-1 indicates that the poorly covered configuration of eight active modules cannot achieve the 0.98 two-year reliability for the control computer assumed in this report. Tables 9-2 and 9-3 show that the required reliability may be achieved for all three configurations with no more than two spares. Tables 9-5 and 9-6 show that the NSFTC having the expected coverage will achieve a five-year reliability meeting the 0.98 requirement, with no more than five spares for the eight module configuration with 0.99 coverage, and three spares for q = 4 and a coverage of 0.98.

In practice, the NSFTC will not be a closed system, i.e., repair will be permitted within the two-year period. As a consequence, the number of spares necessary online can be reduced as a function of the maintenance philosophy used. In any case, comparing the results of this analysis with the results obtained for external sparing of low-reliability components (see the discussion of feasibility analysis) shows a strong advantage to the recommended fault-tolerant design approach.

TABLE 9-1 2-Year Reliability Coverage = 0.95			
No. Spares	q = 2	q = 4	q = 8
0	.900217	.810390	.656732
1	.987794	.968067	.912292
2	.994184	.987242	.968231
3	.994598	.989107	.977301
4	.994623	.989266	.978515
5	.994625	.989278	.978656
6	.994625	.989279	.978671
7	.994625	.989279	.978673
8	.994625	.989279	.978673
9	.994625	.989279	.978673

TABLE 9-2 2-Year Reliability Coverage = 0.98			
No. Spares	q = 2	q = 4	q = 8
0	.900217	.810390	.656732
1	.990559	.973046	.920362
2	.997359	.993451	.979891
3	.997814	.995499	.989847
4	.997843	.995679	.991221
5	.997845	.995693	.991387
6	.997845	.995694	.991405
7	.998845	.995694	.991407
8	.997845	.995694	.991407
9	.997845	.995694	.991407

TABLE 9-3 2-Year Reliability Coverage = 0.99			
No. Spares	q = 2	q = 4	q = 8
0	.900217	.810390	.656732
1	.991481	.974706	.923052
2	.998421	.995529	.98380
3	.998890	.997640	.994066
4	.998920	.997827	.995671
5	.998921	.997843	.995691
6	.998922	.997844	.995693
7	.998922	.997844	.995693
8	.998922	.997844	.995693
9	.998922	.997844	.995695

TABLE 9-4 5-Year Reliability Coverage = 0.95			
No. Spares	q = 2	q = 4	q = 8
0	.768896	.591201	.349518
1	.948781	.867827	.676601
2	.980345	.948723	.848775
3	.985268	.967649	.915910
4	.985987	.971524	.937506
5	.986088	.972249	.943569
6	.996102	.972376	.945105
7	.986104	.972397	.945465
8	.986104	.972401	.945544
9	.986104	.972401	.945560

TABLE 9-5 5-Year Reliability Coverage = 0.98			
No. Spares	q = 2	q = 4	q = 8
0	.768896	.591201	.349518
1	.954462	.876562	.686930
2	.988050	.962649	.870150
3	.993454	.983425	.943848
4	.994629	.987812	.968304
5	.994388	.988660	.975386
6	.994404	.988813	.977238
7	.994406	.988839	.977685
8	.994407	.988844	.977786
9	.994407	.988845	.977808

TABLE 9-6
5-Year Reliability
Coverage = 0.99

No. Spares	q = 2	q = 4	q = 8
0	.768896	.591201	.34951
1	.956355	.879474	.690373
2	.990633	.967327	.877351
3	.996204	.988745	.953328
4	.997053	.993314	.978798
5	.997177	.994206	.986249
6	.997195	.994369	.988217
7	.997197	.994397	.988697
8	.997198	.994402	.988807
9	.997198	.994403	.988831

10. Conclusions and Recommendations

This study has shown that a strong economic and safety incentive exists to use fault-tolerant computers in many applications. The cost of ownership for a computer system is primarily determined by the cost of maintenance and downtime of the system. An analysis of the maintenance costs for simplex and fault tolerant computer systems indicate that fault tolerance can reduce ownership costs by a factor of 60 for large systems. Multiplying this savings by the number of computer systems in operation within the Navy results in a savings of many tens of millions of dollars.

This study also surveyed Navy applications that appear most amenable to the use of fault tolerant computer design. These applications include: avionics, shipboard control, buoys and satellites. These applications vary in computer performance requirements and the quantity of innate "inertia."

Inertia is used here to signify the ability of a system to continue running in an acceptable manner while the computer is offline for repair. In many applications recovery latencies in excess of several seconds is permissible. In other applications, outages of any duration are not permitted. Furthermore, some applications permit the output of erroneous information for short periods of time. Conversely, there are applications that cannot tolerate even a single bad output. These diverse scenarios lead to widely different architectures and recovery algorithms.

Performance and memory requirements for each of these applications also vary widely. In addition, the growing trend in the Navy to use high level languages should be reflected in the hardware specifications. High level languages usage is facilitated when the central processing unit operates on 16-bit or larger words. Concomitant memory size should be in excess of 16K-words. Additional features such as stack handling or special data structure support may be useful for some languages.

Supporting diverse requirements imposed by reliability and performance may be accomplished by developing custom architectures for each application. This approach has limited appeal due to the inherent costs. The approach suggested in this report is to design a standard internal bus configuration, error detection methods, fault recovery methods and input/output interface. Within the broad environment provided by these standard building blocks, the user is free to select the processing element and memory size necessary for an application. The key goal of by the hardware and software for the proposed architecture is to establish a user-transparent mechanism for implementing fault tolerance.

It is recommended that future work concentrate on an analysis of system software requirements and tradeoffs. One of the primary issues not addressed in this report is the binding of operating system functions to processing modules. That is, the NSFTC

consists of intelligent input/output devices as well as the SBP modules. An analysis is required to determine how to partition the system software among these resources.

This study has indicated that a low cost and highly reliable fault tolerant computer may be constructed using off-the-shelf-components. This thesis should be tested by implementing a breadboard version of the NSFTC. FST recommends that a specific application be addressed in order to make the results most meaningful.

FST recommends that the breadboard design begin with a careful analysis of the requirements for the selected application. From these requirements, hardware is chosen and configured. One of the key parameters to be measured by testing is coverage. Consequently, the breadboard should be constructed in a manner suitable for injecting large numbers of faults and measuring the system response.

FST additionally recommends that the breadboard design include a demonstration of a portion of the application selected. This is necessary to provide insight into the ability of the architecture to provide the required performance. Performance measurements can be made by timing the execution of segments of software, estimating the system software overhead and observing bus conflict rates.

Upon completion of the design and testing of the breadboard, it will be possible to refine reliability and performance models for the NSFTC. These models are necessary for predicting the configuration requirements necessary for meeting the needs of other applications. FST recommends that a final goal of future work be to determine a straightforward method for evaluating requirements vis-a-vis numbers of active modules, spare modules, external bus configuration and recovery mechanisms to be employed for achieving characteristics determined by application requirements.

11.0 Bibliography

ARRO80 Arsenalt, J.E. and J.A. Roberts, "Allocation and Estimation," in Reliability and Maintainability of Electronic Systems, J.E. Arsenalt and J.A. Roberts, eds., Computer Science Press, Potomac, Md., 1980, ch. 8

BOSC69 Bouricius, W.G., W.C. Carter and P.R. Schneider, "Reliability Techniques for Self-Repairing Computer Systems," Proc. 24th Nat. Conf. ACM, 1969, pp.295-383

CAWJ71 Carter, W.C., A.B. Wadia and D.C. Jessip, "Implementation of Checkable Acyclic Automata by Morphic Boolean Functions," Symp. Comp. and Automata, Polytechnic Inst. of Brooklyn, April 1971, pp. 465-482

LAMA80 Lamarre, B.G., "Mathematical Modeling," in Reliability and Maintainability of Electronic Systems, J.E. Arsenalt and J.A. Roberts eds., Computer Science Press, Potomac, Md. 1980, ch. 19

NGAV80 Ng, Y.-W. and A. Avizienis, "A Unified Reliability Model for Fault-Tolerant Computers," IEEE Trans. Comp., vol. C-29, no. 11, Nov. 1980, pp. 1002-1011

SIAV81 Sievers, M.W. and A. Avizienis, "Analysis of a Class ot Totally Self-Checking Functions Implemented in a MOS LSI General Logic Structure," Proc. IEEE International Symp. Fault Tolerant Computing, June 1981, pp. 256-261

Appendix I

System Considerations of Fault-Tolerant Computing

I.1 Introduction

Digital systems often experience behavior that is not expected as defined in the original system specification. When these systems are used to protect human life, prevent economic loss, or where repair is costly or dangerous, such errant behavior is especially serious. An automatic maintenance technology called fault-tolerant computing has evolved over the past 25 years to protect computer systems from the effects of failures. Although the technology has been available for some time, its cost made it attractive only to researchers and designers of spacecraft. However, modern technology has made fault tolerant computing cost effective in such tasks as process control, transportation, military and aerospace applications, or wherever maintenance costs are significant.

The effect of fault-tolerant computer design on the overall life-cycle cost of embedded-computer systems may be great in a wide number of applications. Life-cycle costs may be significantly reduced by designing increased testability and maintainability into the computer. Fault-tolerant computer systems can extend periods between scheduled maintenance and postpone corrective maintenance until it can be conveniently performed.

Preventive, or scheduled maintenance procedures are those em-

ployed to reduce the likelihood of system failure whether or not failure has occurred. These procedures generally require extensive system testing and replacing of high failure rate components, whether or not their failure is eminent. Corrective maintenance is the philosophy in which service is initiated after the system has failed. Many systems employ both techniques to increase overall availability.

Automatic maintenance procedures can make a significant impact on reducing cost of preventive and corrective maintenance. This is because automatic maintenance performs corrective procedures without assistance from service personnel. Preventive maintenance costs are also affected since the period between scheduled maintenance is increased. Furthermore, the system itself can assist service personnel during preventive maintenance, thereby reducing the maintenance time.

The starting point for automatic maintenance in embedded computer systems is with the computers in the system. If the reliability of a computer is assured, the computer may be used to execute subsystem testing, fault diagnosis and spare switching. The philosophy espoused by the technique described in this paper is to slightly increase the purchase cost of a system by adding an automatic fault-handling capability in order to greatly reduce maintenance costs during the lifetime of the system. The technique employs redundant components organized in an architecture

that can manage internal faults without human intervention.

I.1.1 Definitions

To begin, a few definitions are in order. Errant computer behavior is typically called a malfunction when it is only temporary and a failure when it is permanent. The result of this unexpected behavior within a digital system is to cause some disruptive change in one or more logic values. Such changes of logic values are termed physical faults if they are caused by a change in the operation of physical components within the system. These faults may result for example, from environmental over-stress (thermal, radiation), logic gate failures or power supply failures. The term man-made fault is used when the unusual behavior was the result of an error committed by the human designer either in the specification of the system or in the translation of that specification into an implementation. Faults typically classified as man-made result, for example, in software "bugs," and logic design errors. The term fault will be used in this discussion to refer to either a physical or man-made fault.

In order to eliminate the confusion between faults, failures and errors, the following tautology is useful. A failure is a physical phenomenon such as a break in a wire. It causes a fault which is the change in value of logic variables at the point of the failure. The fault inserts incorrect information into the computation process and may cause one or more errors depending

upon the location of the failure, the state of the machine and the current input values. Finally, an error is defined as the corruption of any single line logic value.

A fault is termed active if it causes at least one error. During the period that a physical fault is active, the digital system deviates from its program-specified behavior, that is, the state transition function is corrupted. When a man-made fault is active, the operation of the system deviates from the designers expectations,. However, the system is in essence performing "correctly," because it is performing as designed. An active fault causes the system to incorrectly execute a specified information transformation task. Important information stored within the system may be permanently damaged and/or the system may cause serious losses in terms of human safety, revenues or property.

I.1.2 Fault Avoidance and Fault Tolerance

One approach to protecting a digital system against faults is to build the system such that it avoids faults. This technique, called fault avoidance, assumes that it is possible to design out the possibility of faults occurring. Procedures generally employed to obtain a reliable system via this approach require: 1) obtaining the most reliable components available; 2) exhaustively testing these components under the range of conditions expected for the system; 3) using well-matured technologies for

interconnection of components and assemblies; 4) shielding to eliminate the likelihood of external interference; and finally, 5) exhaustive testing of the completed system to verify that it meets all performance and reliability requirements.

This approach is often augmented with a manual repair capability such that it is possible to restore the system to normal operation if it does indeed fail. Maintenance procedures are provided that attempt to isolate the cause of the problem, replace the offending unit and restart the processing task as quickly as possible. The system continues to run until the next fault occurs. In many applications, however, the use of manual repair is unacceptable for at least the following three reasons: 1) excessive delay in restoration of normal service; 2) inaccessibility of certain systems; and 3) excessive cost in both down time and in maintenance.

Fault avoidance is optimistic and overlooks, for example, the fact that real components do fail and that no amount of shielding will protect against all external interference. In fact, shielding has been found to be totally useless in some applications. The best known recent example of this was the shielding that was suggested to protect dynamic memories from the deleterious effects of alpha particle radiation. After much study, it was established that the source of these particles was the integrated circuit package itself! Fault avoidance therefore clearly has

limited applicability, especially for complex systems that are installed in relatively hostile environments and expected to perform without incidence for long periods of time.

An alternative approach to fault avoidance and manual repair is to design a digital system to be fault-tolerant. In this approach, faults are expected to occur and to cause errors. Protection against the occurrence of faults is furnished by special redundancies built into the system. These redundancies provide for automatic detection, isolation and recovery to normal operation, ideally without interruption of normal service. Contrast this technique with the fault avoidance method in which the maintenance procedure is invoked after the system has failed, and the system remains unusable until repair is complete.

Fault-tolerance, therefore, is the unique attribute of a system that allows it to continue its program-specified behavior in spite of the occurrence of faults. As indicated above, implementing fault-tolerance in a system requires the use of special redundancy. This form of redundancy is called protective redundancy. The term implies the use of components that would otherwise be unnecessary in a system in which faults never occurred.

There are three basic forms of protective redundancy that may be incorporated into a digital system. Hardware redundancy involves the use of redundant hardware modules such as processing ele-

ments, switching configurations and detection circuitry[1]. Software redundancy may be used to solve the problems caused by both man-made[2] and physical faults[3]. Finally, time redundancy is repeated execution of a given system state transition.

Fault-tolerance and fault avoidance are complementary techniques that may be used to protect a system against faults. It is often desirable to protect different portions of a system using one or the other method. Experience and analysis are necessary to determine the appropriateness of each method. Fault-tolerance does not eliminate the need for good design or use of the best components. Its real purpose is to provide reliability and availability either less costly or beyond what is possible through the use of fault avoidance techniques.

Fault-tolerance should always be considered an adjective that describes a particular architectural feature of the computer system. Other architectural features such as instruction set, memory configuration, internal communication philosophy, etc., determine the performance characteristics of the system. Therefore, the description of a computer system must include requirements both for reliability and performance. Since the inclusion of fault tolerance into a system requires the addition of extra hardware or software, there is often a trade-off made between an acceptable performance level and a safe reliability level. Performance and reliability goals must be established as early as

possible in the design of a system to guide trade-off decision making.

I.1.3 Implementing Fault Tolerance

Perhaps the first application of fault tolerance can be found in Babbage's Analytical Engine designed in the 1820s. This well-known machine was built such that if any computation was out of specification, gears would jam, indicating an error. In more recent history, starting with the work of Von Neumann[4], a methodology has been evolving for incorporating fault tolerance into a system design. The starting point in the present form of that methodology is to define an architecture and technology that is capable of meeting a set of performance requirements assuming fault-free operation. The specification of the design must be flexible enough to permit an easy inclusion of the extra hardware and software that will be required to provide the fault tolerance of the system. After the system has been so defined, the following series of design steps are performed[1]:

a. The reliability goals of the system must be defined. This includes postulating the fault set that must be tolerated by the system, and quantifying important reliability parameters such as the period of time for which the system must have a given reliability, the mean-time-to-first-failure, and so forth. A method for evaluating actual reliability must also be resolved.

b. Fault-detection algorithms are selected and merged with the

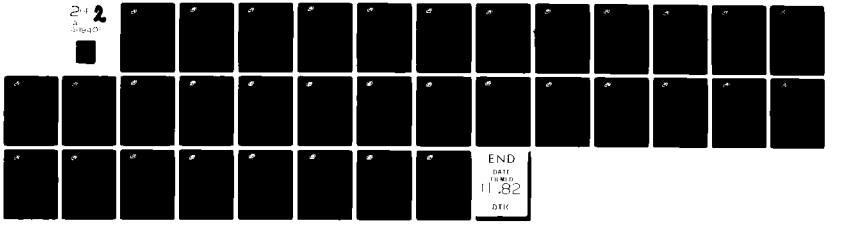
design. Selecting appropriate algorithms and the means to incorporate these with the original design are usually driven by the reliability goals established in Step a. These algorithms may be implemented completely in hardware, completely in software, or in a combination of the two.

c. Fault-recovery algorithms must be chosen. These algorithms are invoked by signals from the fault-detection algorithms. The purpose of these algorithms is to restore the system to normal operation or to deactivate the system in a controlled manner if normal operation is not possible (safe shut-down). Recovery may include: the replacement of failed units with standby spares; forcing a system reset command; or reloading memory. Depending on the application, it may be necessary to provide fault-masking techniques. These techniques are a special case of fault recovery in which the redundant elements hide the effects of a fault. An example of this would be the error correction-capability of Hamming-coded memory.

d. An evaluation of the system must be performed to determine if the reliability goals were satisfied. This evaluation may be done by physical or digital simulation, analytical modeling, or a combination of all three.

e. Design refinement must be done if all reliability goals are not met. It is likely that various subsystems are found to contribute a disproportionate unreliability to the overall system. More effective means will have to be found to

AD-A119 405 FAIL-SAFE TECHNOLOGY CORP LOS ANGELES CA F/G 9/2
U.S. NAVY FAULT-TOLERANT MICROCOMPUTER.(U)
JUL 82 M W SIEVERS, G A KRAVETZ, B A DUSSIA N00014-82-C-0126
UNCLASSIFIED NL



balance these contributions such that the system becomes more reliable.

I.1.4 Introduction to Remaining Sections

The remaining sections of this appendix are devoted to filling in the details for the concepts outlined above. In the next section, physical faults, their classes, metrics and effects on digital system components are discussed. The third section is a presentation of the parameters used to describe and compare fault-tolerant systems. This is followed with a presentation of the various techniques available for detection, isolation and repair of physical faults.

I.2 Physical Faults

Physical faults are caused by many mechanisms ranging from, for example, minute changes in semiconductor devices to more spectacular causes such as lightning strikes. Although it is not practical to enumerate all sources of physical faults, these faults can be classified into groups according to their manifestation and range of damage in a circuit function.

I.2.1 Classification of Physical Faults

Three classes of mechanisms are responsible for physical faults. Permanent hardware failures are those phenomena that result in irreversible damage to a hardware component. Temporary malfunctions of components may result from, for example, temperature

variations, or power supply drift. The last class of phenomena affecting digital systems are those faults induced by external interference. Examples of these mechanisms are electro-magnetic interference and radiation.

The determination of which class of phenomena a particular fault belongs is largely a function of three parameters. The duration of a fault distinguishes between permanent and transient faults. The extent of a fault is the "distance" over which it is causing damage, that is, a fault may be either local or distributed. Finally, the value of a fault refers to whether or not the fault is determinate or indeterminate.

Most often, it is not possible to immediately distinguish between permanent and transient faults. A transient fault may be caused either by temporary component malfunction or by temporary external interference. In most systems, the transient fault rate is one or two orders of magnitude greater than the permanent failure rate. That is, the probability of a permanent failure is predicted from a different arrival rate and model than that used for transient faults. Typically, a duration parameter is also employed as a classifier for transient and permanent faults. When the duration of a fault is longer than the duration parameter, it is classified as a permanent fault by recovery routines.

The extent parameter applies to both transient and permanent faults. It is used to quantify the number of logic variables

that are simultaneously damaged by a single mechanism. When multiple variables are affected, the fault is termed distributed. If the fault damages only a single variable, the term local is used.

The value of a fault is called determinate when the affected variables are forced to a constant value during the entire duration of the fault. Faults classified as determinate are the well-known stuck-at-one and stuck-at-zero faults. Indeterminate faults, also called stuck-on-x faults, are those faults that vary between "1" and "0" during the fault duration. These faults may be the result for example, of shorting (so-called bridging faults), or parasitic circuit elements[5,10].

Faults may be further classified as one-use and repeated-use faults. A fault that is used only once prior to detection is called a one-use fault. In iterative algorithms such as multiplication, various digit circuits are used repeatedly to form the final result. Any faults affecting this circuitry will result in so-called repeated-use faults.

I.2.2 Effects of Faults on Arithmetic Elements

The manifestation of a fault in arithmetic computations can be determined in a straightforward manner if deterministic faults are assumed[6]. There are eight basic arithmetic operations performed in a typical computer system. These are: 1) transfer,

2)k-digit left shift, 3) k-digit right shift, 4) k-digit range extension, 5) k-digit range contraction, 6) modulo M addition or subtraction, 7) additive inverse and 8) k-digit roundoff.

Before embarking into this analysis, it is appropriate to introduce a few more concepts relevant to arithmetic errors. A fault affecting an arithmetic unit will transform an n-digit radix r perfect result $s=(s_{n-1}, s_{n-2}, \ldots, s_1, s_0)$ to the observed result s^*. The observed result differs from the perfect result in at least one digit. The error number $e=(e_{n-1}, e_{n-2}, \ldots, e_1, e_0)$ defines the digit-wise difference as $e_i = s_i^* - s_i$ for $i=[0, n-1]$. Clearly the digit values in e will be in the range $[-r+1, r-1]$. The number represented by the digit-vector e is called the error value E, which is computed as the sum from 0 to n-1 $r^i e_i$. Thus E will be in the interval $[-(r^n - 1), (r^n - 1)]$. The magnitude of E, $|E|$ is defined as the absolute value of E. When e is recoded to have the minimum of non-zero digits, this minimum number is called the arithmetic distance between s and s^*, or arithmetic weight of e.

This analysis assumes that all arithmetic primitives are computed in parallel. This implies that all effects are caused by single-use faults. Furthermore, the effects of faults will be considered only for the binary range complement number representation system.

1) Data transfer:

The fault-free transfer of an n-bit vector implements the equation A = B where A and B are the values represented by the contents of two registers a and b. A fault affecting location i of such a transfer results in |E| of 2^i and an error weight of 1.

2) k-digit left shift:

In both arithmetic and logical k-digit left shifts, assuming that overflow or faults do not occur, the operation is to multiply the value represented by the contents of a register by 2^k. The well-known implementation of this multiplication is to shift zeros in at the least significant end of a shift register and to move all bits within the shift register to the left k positions. The most significant k bits are discarded. The inability to correctly shift zeros into the register will yield an |E| of $\sum_{i=0}^{k-1} 2^i$. In general, a fault in location j, that is, a fault that results in the inability to either correctly read the contents of the j-th cell or the inability to correctly set/reset the j+1 cell. Error magnitude in the general case is a function of the contents of the register.

3) k-digit right shift:

The arithmetic right shift is performed by copying the contents of the n-1st cell into the k cells n-2 to n-2-k+1. The contents of the cells starting at n-2 are moved left k

positions and the contents of the least significant k bits are discarded. A fault affecting x_{n-1} or its set/reset circuitry will result in errors in the k+1 leftmost cells. The associated |E| is $\sum_{i=n-1-k}^{n-1} 2^i$. When a fault affects the j-th cell or the set/reset of the $j-1^{st}$ cells the error magnitude depends on the register contents. In a k-digit logical right shift, k zeros are shifted into the most significant bit and the contents of each cell are moved k positions to the right. Damage to this algorithm is identical to that observed for an arithmetic shift.

4) Range extension and contraction:

In the k-digit range extension algorithm, k digits equal to the most significant bit x_{n-1} are concatenated to the left of x_{n-1} to create an n+k digit vector. An incorrect value of x_{n-1} results in an |E| of $\sum_{i=n-1}^{n-1+k} 2^i$. A fault affecting the circuitry used to sense the value of x_{n-1} or the copying of that value results in an error magnitude of $\sum_{i=n}^{n-1+k} 2^i$. If only the j-th digit for $n-1<j<n+k$ is affected, the |E| is 2^j.

The k-digit range contraction algorithm is simply the inverse of the range extension algorithm. That is, the k leftmost digits (x_{n-1}, ... , x_{n-k}) are removed when these are equal to x_{n-k-1}. An incorrect removal yields an |E| of $c2^{n-k}$ where $1 \leq c \leq 2^k - 1$. An incorrect value of x_{n-k-1} (e.g., a 0) will cause the removal of the k most significant

digits (e.g., all ones) resulting in an |E| of $\sum_{i=n-1-k}^{n-1} 2^i$.

5)Modulo M addition or subtraction:

The modulo operation requires that M or -M be "cast out" from the sum or difference respectively. A fault in bit location j will cause a local error magnitude of 2^j. However, an additional error value may occur if the local fault either promotes or inhibits the casting out process. In this case, the local fault results in an |E| of $M-2^j$.

6) Additive inverse:

The two's complement additive inverse is computed by first complementing each bit in the n-bit word, then adding 1 to that word. A fault in the negation process results in a local error only and yields an |E| of 2^j if the j-th bit is affected. The addition process is affected as described above.

7) k-digit roundoff:

There are three basic roundoff techniques used in digital systems: a) the value of the k digits $x_{k-1}, \ldots, x_0$ is tested followed by the addition of either 0 or 1 to x_k, b) always set x_k to a 1 and c) truncation (no arithmetic performed). In cases a and b the error magnitude for a fault either in setting/resetting of the k-th digit or in testing the range of the lower significant digits will result in an |E| of 2^k. Case a is also susceptible to the

addition errors.

I.2.3 Physical Fault Effects on Control Primitives

Typical primitives found in control circuitry are combinational elements such as gates, programmed logic arrays, comparitors, multiplexers, demultiplexers and read only memory, as well as small sequential primitives such as registers and counters. These elements are used to implement the state transition function of a finite-state automaton. As such it can be expected that any faults affecting the control mechanism will result in an altered transition sequence. Depending upon the implementation, the faulty control unit may halt, loop on a set of states or follow some unknown transition sequence.

An indeterminate fault called a parasitic flip-flop fault has been found to cause very unusual behavior[5,10]. Several mechanisms have been found that actually add additional states. For example, combinational circuitry is transformed into a sequential network due to the fault. This type of fault will generally result in one of the three symptoms indicated above.

I.2.4 Memory Faults

Memory, whether it is core or semiconductor RAM contains data input and output conditioning circuitry, address circuitry and control circuitry. Data conditioning includes sensing and amplifying functions. Faults affecting these functions result in read or write errors. Depending on the actual mechanism involved, the

fault may be distributed in nature. Clearly faults affecting address circuitry will produce accessing errors. In read/write memory there may be no noticeable effect of such faults. For example, if a fault consistently causes the access of the wrong memory location, but there is still a unique location for each address, then there will be no detected error. On the other hand, address errors resulting in fewer accessible memory locations than expected will produce data errors. Bridging affecting both data and address wires will yield corrupted data. Finally, control circuit problems will prevent correctly reading or writing memory.

The memory plane itself is susceptible to the expected deterministic faults resulting in data errors. Dynamic memory is often plagued by transient errors resulting in loss of data. However, in these cases it is possible to restore faulted memory locations by rewriting the lost information. Latch-up of individual memory cells may occur in certain implementations. In latch-up, a cell appears to be stuck at a particular logic value. Upon power cycling, the latch-up condition disappears.

Finally, one of the more insidious RAM problems is the result of pattern sensitivity. Due the layout of particular RAM chips, a strong capacitive coupling exists between adjacent cells. Certain data patterns present in those coupled cells make setting or resetting particular bits impossible. Again, the result is

apparently a transient fault. Although tests for pattern sensitivity exist, these tests are often very time consuming and by necessity make certain assumptions regarding adjacency of cells. As such, a test generated for one memory chip is not necessarily valuable for another.

I.3 Metrics

Having outlined the classes of physical faults, we now look at the metrics of reliability. These metrics are used to establish quantitative goals for the digital system. Furthermore, a system design can be tested and the metrics computed to determined whether or not the system meets the established requirements.

I.3.1 Basic Notions

The reliability of a system is defined as the probability that the system is operating correctly up to the time t=T, given that the system was operating correctly at the starting time t=0. In most cases, the starting time is defined as the last time the system was checked and found to be operating correctly. Reliability is an attempt to predict the future behavior of a system and is affected by four factors[7]:

a. Hardware reliability is the reliability of the physical components that make up a system. Prior to extensive accumulation of statistics on the causes of system down-time, it was thought that hardware failure was the most likely culprit. More recently, however, it has been determined that only about 20% of all system down-time can be attributed to hardware failure.

b. Software reliability includes all programming errors that result either in the destruction of stored data or in program loops that can only be exited by re-initialization of the system. In a mature system, on the order of 15% of all downtime results from software errors.

c. Recovery problems are difficulties encountered in attempting to restore a failed system to normal operation. In general, recovery requires four complex steps: 1) fault detection, 2) fault isolation (diagnosis), 3) repair and d) re-initialization of the system. Minimizing this contribution to downtime requires an efficient and highly effective maintenance and diagnosis methodology. Problems arising in recovery account for 35% of the downtime in conventional (manually maintained) systems.

d. Procedural errors cause the remaining 30% of system downtime. These are errors committed by system operator or maintenance personnel. Frequently, the ultimate cause of procedural errors is poor or incorrect documentation. From a practical point of view, it is impossible to reduce the probability of a human mistake to an acceptable level without reducing the need for manual intervention in the system operation. Additional reliability can be gained by providing a system attribute for checking those human activities that cannot be dispensed with.

Availability of a system is defined as the probability that the system is operating in a satisfactory manner at any point in time. One of the goals of the system engineer is to determine what a "satisfactory manner" actually means. As an example, it might be desired to provide very high availability but only average reliability for a given system. Mathematically, availability is (total up-time)/(total up-time + total down-time). Availability is affected both by system reliability and a term called mean-time-to-repair (MTTR). MTTR is essentially the time taken by the recovery process.

One of the more commonly used metrics to measure the quality of a

digital system is the mean-time-to-failure (MTTF). This is defined as $\int_0^\infty R(t)\,dt$. For a fixed "mission time" T, it is necessary only to compute the values of $R_A(T)$ and $R_B(T)$ to compare two systems A and B. The reliability improvement factor (RIF) is defined as RIF = $(1-R_A(T))/(1-R_B(T))$. This parameter measures the improvement in using system B over system A. When a mission time is not specified, it is often useful to compute the mission-time-improvement-factor (MTIF). MTIF is defined as $(T_B)/(T_A)$ where T_A, T_B are the times at which system A and B respectively fall below a minimum specified reliability.

I.3.2 Coverage

Coverage, c, is an extremely significant parameter affecting system reliability. It was defined in a fundamental paper[11] as the conditional probability that a system can recover given that the system fails. The definition of exactly what recovery entails is decided by the system designer.

Reliability is proportional to a geometric series in c, i.e., $R = a_0c^0 + a_1c^1 + \ldots + a_sc^t$. The model on which this equation is based assumes that there are t spare units that may be switched into a system to replace a failed unit. The coefficients a_i are a function of the failure rate of powered and unpowered spares, the number of identical modules operating simultaneously and the number of single failures that can be tolerated before the system fails.

As an indication of the importance of coverage, in a particular implementation, assume that an infinite number of spare units are provided. A coverage of 0.7 will only yield an MTIF of three over a non-redundant system. On the other hand, a coverage of 0.99 is capable of producing an MTIF of close to 70. In this example, it is clear that it is more advantageous to improve coverage than to add more spares to a poorly covered system.

Coverage is a very difficult parameter to measure. In a strict sense, the determination of coverage requires the application of every postulated fault to every possible location in the system and observing whether or not recovery was successful. For any non-trivial system this is clearly impossible. Thus coverage is typically measured via analysis and judicious application of digital and physical simulation[8].

I.4 Implementation of Fault Tolerance

The key to successfully creating a fault-tolerant system is to provide protective redundancy. As mentioned in the first section, there are three basic ways in which this redundancy can be provided: 1) hardware, 2) software and 3) time. These redundancies are used to implement detection algorithms as indicated in the last section. After detection of a fault, it is necessary to isolate the fault location via diagnostic algorithms. This is followed by repair of the system through use of spare

units. In this section, we survey testing, diagnosis and repair.

I.4.1 Testing for Physical Faults

Fault detection is the determination that the digital system has suffered from a malfunction. Obviously, this determination is required prior to invoking the recovery algorithms which diagnose the location of the fault and effect appropriate restoration procedures. As such, detection is the cornerstone of the recovery process. In this discussion, it will be assumed that a fault has been detected, or a false alarm has occurred, when the detection mechanism issues a fault signal. A false alarm is the incorrect assertion of the fault signal due to some malfunction in the detection mechanism.

Fault detection is implemented via a combination of hardware, software and time repetition methods that result in the assertion of the fault signal. These methods are often grouped according to their intended use. Initial testing is performed prior to normal system operation. It is used by a manufacturer before final system integration to establish that the fabrication and assembly processes have not produced defective hardware. Concurrent or online testing is performed during normal system operation. This form of testing primarily looks for faults that have appeared after the system is installed. By its nature, concurrent testing is useful for detection of short duration transient faults. Scheduled or offline testing requires that normal system opera-

tion be terminated so that special testing procedures can be invoked. This type of testing usually is less costly in hardware terms but suffers from its inability to detect short duration transient faults. More importantly, online techniques have an advantage over offline techniques in that these are capable of detecting faults sufficiently early to protect against massive disruption of operation or irreparable data contamination. Finally, redundancy testing is special testing performed to verify that the fault tolerance features of the system will function correctly when called upon.

Hardware techniques for online testing are probably the best known methods, having been used since the earliest digital systems. Typical techniques include error coding (parity, Hamming, checksums, and so forth), self-checking circuits, duplication and comparison, majority voters with disagreement detectors, watchdog timers, completion signals and special circuitry to monitor critical elements such as the power supplies and clocks.

Software may be employed to implement both online and offline testing. Online software testing is implemented through concurrent execution of multiple copies of a program, or using special checking procedures built into the software itself. When multiple programs are used, these programs store results or summaries of results in shared memory locations. Each program can retrieve the information deposited by other programs. A software-imple-

mented comparison is then performed. The other software technique makes use of special monitoring software, running in a dedicated subsystem. This software observes the behavior of the remainder of the system.

As can be expected, software fault-tolerance techniques are not as prompt in detecting faults as hardware techniques. Furthermore, by its nature, software fault tolerance is susceptible to the faults that it is intended to protect against. However, it has the advantage of being relatively easily installed in existing systems.

I.4.2 Diagnosis

The purpose of fault diagnosis is to locate the source of a fault to the least replaceable unit. This might be a component, circuit or subsystem. In general, diagnostic algorithms make use of the detection hardware and special test sequences. In some cases, the detection hardware may point directly to the faulty unit, thus no special additional isolation is required. On the other hand, it is possible that detection will indicate one of several probable failed units. In this case diagnostic routines are necessary to find the most likely culprit.

As an example, consider two non-redundant processors executing the same program with a comparator checking their outputs. When a disagreement occurs, it is not possible to determine the source of the fault. Typically, in order to minimize the disturbance

caused by the detected error, a quick diagnostic program is executed. The purpose of this test is to isolate to the failed processor such that this processor can be taken offline while the other continues operating. Thus this test need not make a detailed analysis of the component or subsystem within the processor that is faulty. This makes a fast evaluation possible. Once the faulty processor is disconnected, offline diagnostics can be run to isolate to the least replaceable unit to effect repair.

On the other hand, if the processors were designed to be totally self-checking, then each processor independently determines its own ability to perform correctly. In this case, first level diagnosis is done in hardware, and no additional analysis is required. As before, once the faulty processor is taken offline, additional diagnostics may be run to find the faulty replaceable unit.

Another use for diagnostic routines is to verify that repair is successful. That is, detailed offline diagnostics are used to pinpoint an error source such that it may be replaced to effect repair. However, due to coupling within a system it may only be possible to determine the most likely module to replace. When it is not possible to find a fault source with 100% accuracy (and this level of accuracy is rare) it is necessary to verify that the correct replacement candidate was specified. This is best done by rerunning the diagnostic programs after repair to deter-

mine whether or not the fault indication recurs. Clearly, if it does, then either the wrong module was selected for replacement, or the inserted module was also faulty. This latter possibility can be reduced if the spare pool is frequently subjected to tests.

I.4.2.1 "Start Small" Fault Diagnosis

One of the most common methods of performing diagnosis is called the "start small" technique. In this approach, the hardcore of the system is tested first followed by tests of the rest of the system over paths already tested. That is, a small portion of the hardcore is verified first. Once this appears to be operational, it is used to verify more of the hardcore. This in turn is used to check still more hardware, until the entire system is checked. The hardcore is defined as the portion of the hardware that must be operational in order that the rest of the testing can be performed. Typically, the hardcore consists of some portion of the central processing unit, for example, the micro-instruction sequencer, and perhaps the arithmetic logic unit. The hardcore is usually verified either by automatic techniques, for example, an independent co-processor that exercises its circuitry, or by manual procedures implemented by maintenance personnel.

In a simplex (non-redundant) system, the only option available for checking the hardcore is via manual procedures, because there

is no independent processor available to carry out this job. This is an ad hoc approach and, of course, suffers from the possibility of human error. Since it is generally costly to keep a system offline, it is desirable to repair that system as quickly as possible. The implication here is that it is important to minimize mistakes in the repair process. This implies the use of systematic procedures that require little human involvement.

A goal in the design of a system is to minimize the size of the hardware within the hardcore while providing automatic means to check this circuitry. The size of the hardcore is, to a large degree, a function of where diagnostics are stored. When diagnostics are stored in offline media such as tape or disk, the hardcore must include these devices, their interfaces, and all software/firmware necessary to accomplish data transfers.

Some systems are designed with a portion of the diagnostic software within ROM. The purpose of this software is to check the basic processor function. If the processor appears operational, bootstrap loaders are utilized to call in diagnostics from offline storage. In this situation, the hardcore does not include the more fault-prone offline storage units, and thus there is a greater probability that the diagnostic process will be capable of executing.

A typical diagnostic sequence would begin by testing the system

clock. This can be done either by maintenance personnel or by special-purpose hardware. By its very nature, the clock cannot be checked by diagnostic routines running within the processor being checked. If the clock is functional, the instruction fetch and instruction cycle firmware can be checked. This type of checking is best done at the micro-instruction level and is done to ensure that the processor is capable of fetching and executing instructions. ROM bootstrap loader diagnostics, instruction decoding circuitry, data paths, data registers, the ALU, and input/output control logic are then partially tested in preparation for loading subsequent diagnostics for the rest of the system. If no problems have been detected to this point, a portion of memory that will hold diagnostics loaded from offline devices is tested. This will be followed by loading diagnostic routines from tape or disk into tested memory. At this point, it is possible to complete the checkout of the data paths, data registers, ALU, input/output control and memory. Finally the input/output devices themselves are checked.

I.4.2.2 Microprogram Diagnostics

In microprogrammed processors, it is possible, and often desirable, to perform diagnosis at the micro-code level. Since micro-code controls the elementary data paths and operations, diagnosis at this level has much more visibility into the system than macro-code would. Often micro-code diagnostics are coupled with special test points inserted into the hardware that can be

selected by the condition code multiplexer typically found in microprogrammed architectures. Thus the micro-code can stimulate certain paths or operations and can monitor the effects of this stimulus via the test points. Scanning procedures[12] may be used to reduce the number of parallel data paths required for accessing test points.

Micro-code diagnostics need not be permanently installed in the control store of the processor. For example, it is possible that maintenance personnel substitute a diagnostic control store for some or all of the control store. That is, the ROM containing micro-code that is normally used to provide processor sequencing is replaced with ROM containing diagnostic routines. In this way, it is not necessary for the seldom used diagnostic ROM to occupy valuable space within the processor. It is also likely that the size of the diagnostic ROM is much greater than the micro-address space available. In this case, the diagnostic ROM can be partitioned into segments, each running to completion before the next segment is inserted. Furthermore, since this ROM can be carried from system to system, it need not be replicated for every installation.

Many micro-coded systems contain a writable control store (WCS). This is a portion of the control store implemented in RAM. When a WCS is available, it can also be used to run diagnostics. These diagnostics can be loaded from main memory into the WCS. A

variant of this approach is to provide a special processor mode in which the data read from main memory are interpreted as micro-code directly. In both cases, the object is to provide a flexible method for executing micro-level diagnostics.

I.4.3 Recovery Mechanisms

Recovery is the restoration of a system to normal (possibly degraded) operation. In the event that normal service is not possible, recovery may be a systematic shutdown of the system. Recovery algorithms can be classified as either being manual or automatic. Manual recovery requires some level of human intervention. In automatic recovery, there is no need for human assistance; the system repairs itself. This discussion will be confined to automatic recovery procedures. Reliability expressions for some of these procedures will be given in the sixth section.

Automatic recovery can be further subdivided into three classes as a function of the state of the system after recovery is completed. Full recovery implies that the system has been returned to full normal operation. That is, the recovered system has all of the processing power and memory size as before a fault occurred. This implies that all damaged information is restored to its fault-free value. In degraded recovery, also known as "graceful degradation", or "fail-soft," the system returns to a fault-free state but has lost some of its processing or memory

capacity. Further, some information may be permanently lost, or some functions may not be capable of complete operation. Finally, safe-shutdown is the limiting example of degraded recovery. This type of recovery is performed when there are no more spare units to switch in for failed units and it is not possible to continue operation above some acceptable level. In this situation, the system shuts down in such a manner that as much data as possible can be saved, and interaction with other systems ceases.

Automatic recovery can also be classified as being static or dynamic. Static mechanisms are those techniques that attempt to hide the effects of a fault through masking. Dynamic recovery implies the substitution of spare units for failed units. Systems can of course combine static and dynamic techniques as suit the reliability and availability requirements.

Finally, automatic recovery can be carried out by both hardware and software methods. In general, the software techniques require more time to complete recovery than hardware methods. Further, software-implemented recovery is susceptible to damage by the fault from which it is supposed to recover. On the other hand, software techniques require little or no additional hardware, so are "cheaper" from the point of view of component cost. Nevertheless, it should always be recognized that software development, in general, is much more costly in terms of manpower than hardware.

I.4.3.1 Masking

The purpose of fault masking is to completely contain the effects of a fault within a module. That is, a module protected by fault masking will not show any outward appearance of having failed. Of course this behavior makes the assumption that the module has not used up its component of redundant submodules. Another feature of this form of redundancy is that there is no need to provide special testing or diagnostic features. That is, testing, diagnosis, and repair are all handled in masking by the same mechanism, automatically and instantaneously. Intrinsic to this feature, then, is the notion that spare units always be powered and ready to be used when called upon. Masking is implemented by error-correcting codes, replicated hardware and redundant software.

One of the most fundamental questions in the use of masking is the size of the protected module. Protection can be performed at the component level, gate level, circuit level, subsystem level or system level. Each level has an associated cost/benefit. However, in all cases, the assumption upon which the use of masking is based is that there is virtually no chance of correlated faults occurring in the redundant module that will cause the module to fail. This implies, for example, that masking would probably not be feasible if employed to protect an integrated circuit package. This is because the chances are good

that if one component in the masking scheme has failed, another will also be faulty in a similar manner.

Masking at the device level can be illustrated by the diode configuration shown in Figure I-1. Assume that the predominant diode failure mechanisms are opens and shorts. In the figure, diodes are arranged in a series parallel configuration such that no single diode failure will cause either an open or short between node X and Y. Of course, the cost of this implementation is four diodes instead of one.

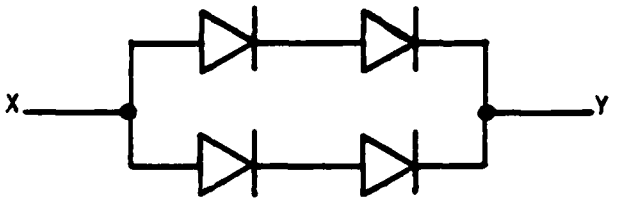


FIGURE I-1 DIODE MASKING

A clever scheme was devised many years ago to protect combinational as well as sequential networks via gate level masking. This method called quadded logic[9] was discovered to be effective when gates occupied different packages, again to reduce the probability of correlated faults. The fundamental ideas behind quadded logic operation is that logic network is copied four times, an error resulting from a faulty element within the network is corrected downstream from the source of the error, and the means for providing this correction are derived from fault-free neighboring signals.

Consider Figure I-2 which depicts a technique that implements masking by providing identical modules M_i and a voting circuit V. The circuit V takes the majority vote of the outputs of the three modules. This technique is called triple modular redundancy (TMR). As long as a fault affects only a single module, the voter will produce the correct result. The assumption here, of course, is that the voter itself is not damaged by a fault. This assumption is usually valid because the complexity of the voter is much less than that of the modules that it is voting on.

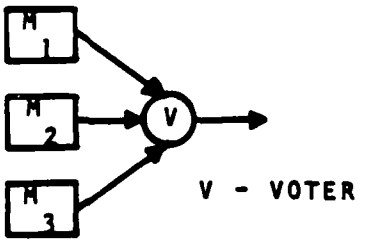


FIGURE I-2 TMR CONFIGURATION

The technique can be generalized to more modules and is called N modular redundant (NMR). For example if there are five modules in the system, it will be possible to tolerate two bad modules. That is, the number of bad modules tolerated by an NMR configuration will be (N-1)/2. The only requirement is that N be odd such that a majority exists.

The concept of NMR masking has been extended to software. So-called N-version programming[2] makes use of multiple concurrently executing programs. These programs are specified, designed, coded, tested, and maintained independently. The system in which N-version programming is included is termed an N-version software system. This system contains special voting procedures and drivers that implement exchange of program outputs and majority voting. If the redundant programs are executed as well on independent processors, then N-version programming has the ability to mask the effects of hardware faults. Perhaps even more important, however, is that since the programs are independently generated, the probability that two contain the same software "bug" in the same segment is very small. Thus N-version programming can also mask the effects of software failure.

Mechanizing N-version programming requires three basic functions. Since each version is independently derived, it can be expected that each will arrive at a particular result at different times. Thus it is necessary that a synchronizing procedure be executed

such that voting is done when it is meaningful. Once it has been determined that information to be voted on is all available, that information must be transferred from the domain of each program to the domain of the voter or voters. If programs are all executing on the same processor, this exchange can be accomplished via passing of pointers. When separate processors are involved, relevant data can be stored in shared memory if this is available. Alternatively, it is necessary for each processor to actually transmit information either via a bus or an input/output interface.

I.4.3.2 Online Redundancy

Online redundancy requires that spare units be switched in for faulty units. Hardware spares may be left in powered off state until necessary or may be left powered on in a stand-by mode. Software module sparing can also be employed. Often it is necessary to bring the switched in spare "up to date" with respect to the current machine state and process.

Software sparing is exemplified by a technique called recovery blocks. In this approach, the software system is structured into blocks. Each block contains a conventional (non-redundant) block which is provided with a means to do error detection. In addition, there can be one or more spare modules. The testing, called acceptance testing is critical to the recovery block methodology. When this test indicates a problem in the block being

executed, another block is selected and run. It is possible that the recovery block technique can provide protection against software errors by requiring that each block be independently coded.

The critical component in hardware sparing is the switching mechanism. After a module has been diagnosed as being faulty, the switching mechanism is responsible for disconnecting the failed unit from the system, and substituting for it a unit from the spare pool. Clearly the inability of the switch to function correctly makes recovery impossible, that is, adversely affects coverage.

Spare switching can be done in at least two ways. The first requires removal of power from the spare unit and application of power to its replacement. In such a system interface logic must be designed such that removal of power renders them incapable of sending a signal. Furthermore, the power switch must be so designed that when instructed to be turned off, there is virtually zero probability that it fails in the on position. Similarly, this switch must turn on with very high probability when instructed to. These switches frequently contain dual coil relays that are configured in a series-parallel masking configuration.

The other approach is to actually disconnect the outputs of the failed unit and connect up those of the spare. The technique, commonly called cross-strapping also makes use of ultra-high

reliability switches in a masking configuration. These switches can be dual-coil relays or special networks of integrated circuit multiplexers.

Beyond consideration of switch implementation, hardware sparing may be used in different architectures. Figure I-3 shows the basic configuration of a hybrid TMR system. This system is a combination of masking TMR with sparing. A disagreement detector compares the voter output with the output of each of the modules in the active configuration. When a disagreement occurs, the detector flags the corresponding unit as having failed. This unit is then replaced by a spare. Generalizing, it is possible to configure a hybrid NMR system.

Another possible architecture is shown in Figure I-4. Here, each module is internally totally self-checking. Thus the failure of any module is recorded by the module itself. Through a query mechanism, surrounding modules can determine when a module has failed. These modules can then agree to disconnect the faulty module from the system and bring up a spare module. Alternatively, the remaining modules may assume the processing tasks of the failed one.

Failed active modules are replaced as long as a pool of spares exist. When the pool is depleted, some systems may continue to operate in a degraded mode. In a degraded mode, the faulty unit

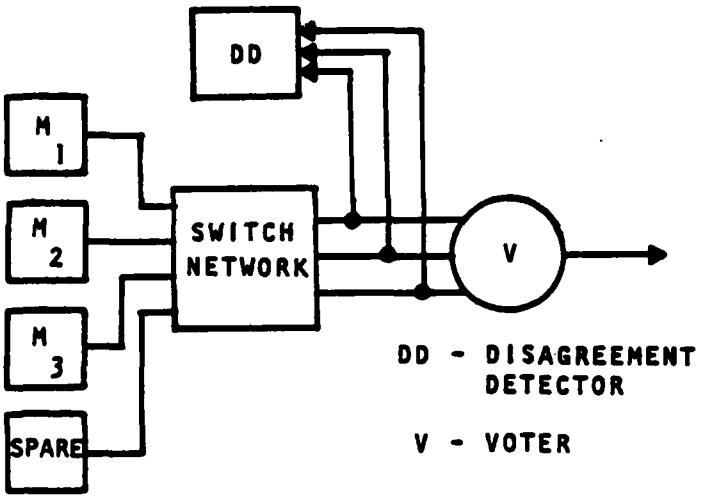


FIGURE I-3 HYBRID TMR

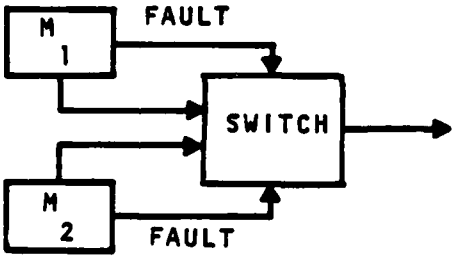


FIGURE I-4 TSC MODULES

is switched offline, but since there is no replacement for it, the system must run without it. Typically, there will be a degradation of performance, memory, bus bandwidth, and so forth. However, this system is otherwise running normally. Eventually, enough modules have failed that the system can no longer provide useful service. At this point, the primary function of the surviving circuitry is to shut down the system in a safe manner.

I.5 Conclusions

Fault tolerance is an attribute of a computer system that permits that system to continue operation in the presence of faults. The implementation of fault tolerance must take into account many parameters, including: initial cost, life-cycle cost, performance, reliability, availability, fault set, response to faults and user expectations. Each application of fault tolerance is unique: there is no such thing as a general purpose fault-tolerant computer system. For example, in one application, it may be permissible to temporarily remove the faulted computer from service during the repair process. In another application, this may result in a disaster, necessitating the use of fault masking techniques. These different scenarios significantly impact the design decisions, cost, reliability and other aspects of the computer system. Consequently, it is necessary to carefully analyze each application to determine the most appropriate design philosophy.

I.6 References

1. Avizienis, A., "Fault-Tolerance: The Survival Attribute of Digital Systems," Proc. IEEE, vol. 66, no. 10, Oct. 1978, pp. 1109-1125

2. Chen, L. and A. Avizienis, "N-Version Programming: A Fault-Tolerance Approach to Reliable Software," Proc. IEEE International Symp. Fault Tolerant Computing, June, 1978, pp. 3-9

3. Wensley, J., et. al., "SIFT: The Design and Analysis of a Fault Tolerant Computer for Aircraft Control," Proc. IEEE, Oct. 1978, pp. 1240-1255

4. Von Neuman, J., "Probabilistic Logics and the Synthesis of Reliable Organisms from Unreliable Components," Automata Studies, C.E. Shannon and J. McCarthy, Eds., Princeton, N.J., Princeton University Press, Ann. of Math. Studies, No. 34, 1956, pp. 43-98

5. Wadsack, R.L., "Fault Modeling and Simulation of CMOS and MOS Integrated Circuits," Bell System Tech. Journal, vol. 57, no. 5, May-June 1978, pp. 1449-1473

6. Avizienis, A., "Arithmetic Error Codes: Cost and Effectiveness Studies for Application in Digital System Design," IEEE Trans. Comp. vol. C20, no. 11, Nov. 1971, pp. 1322-1331

7. Kraft, G. D., and W. Toy, Microprogrammed Control and Reliable Design of Small Computers, Prentice-Hall, New Jersey, 1981

8. Avizienis, A., and D. Rennels, "Fault-Tolerance Experiments with the JPL STAR Computer," Digest of Papers COMPCON72, Sept. 1972, pp. 321-324

9. Tryon, J.G., "Quadded Logic," in Redundancy Techniques for Computing Systems, R.H. Wilcox and W.C. Mann Eds., Spartan, New York, 1962, pp. 205-228

10. Sievers, M.W., and A. Avizienis, "Analysis of a Class Totally Self-Checking Functions Implemented in a MOS LSI General Logic Structure," Proc. IEEE International Symp. on Fault Tolerant Comp., June 1981, pp. 256-261

11. Bouricius, W.G., Carter, W.C., and Schneider, P.R., "Reliability Techniques for Self-Repairing Computer Systems," Proc. 24th Nat. Conf. ACM, 1969, pp. 295-383

12. Eichelberger, E.B., and T.W. Williams, "A Logic Design Structure for LSI Testibility," Proc. 14th Des. Auto. Conf., 1977, pp. 462-468


LMED
8